Microsoft® QuickBASIC

Programmieren in BASIC

Microsoft®

Microsoft® QuickBASIC Programmieren in BASIC

Version 4.5

Für IBM® Personal Computer
Und Kompatible

Microsoft Corporation

Die in diesem Handbuch enthaltenen Angaben sind ohne Gewähr und können ohne weitere Benachrichtigung geändert werden. Die Microsoft Corporation geht hiermit keinerlei Verpflichtungen ein. Die in diesem Handbuch beschriebene Software wird auf Basis eines Lizenzvertrages oder als Einmallizenz (mit der Verpflichtung, sie nicht weiterzugeben) geliefert. Die Software darf nur in Übereinstimmung mit den vertraglichen Bestimmungen verwendet bzw. kopiert werden. Teile dieses Handbuches und/oder der Datenbanken dürfen weder auf elektronische noch mechanische Weise, einschließlich Fotokopien und sonstiger Aufzeichnungen, ohne die schriftliche Genehmigung der Microsoft Corporation vervielfältigt oder übertragen werden.

Microsoft®, MS®, MS-DOS®, CodeView®, GW-BASIC® und XENIX® sind eingetragene Warenzeichen der Microsoft Corporation.

Hayes® ist ein eingetragenes Warenzeichen der Hayes Microcomputer Products, Inc.

IBM® und PS/2® sind eingetragene Warenzeichen der International Business Machines Corporation.

Intel® ist ein eingetragenes Warenzeichen der Intel Corporation.

ProKey™ ist ein Warenzeichen der RoseSoft, Inc.

SideKick® und SuperKey® sind eingetragene Warenzeichen der Borland International, Inc.

WordStar® ist ein eingetragenes Warenzeichen der MicroPro International Corporation.

Printed in Ireland: 06

Artikelnr. 06027

Dokumentnr. 8633-450-00-2

Inhaltsverzeichnis - Übersicht

Einleitung

1. Teil: Ausgewählte Programmierthemen

1. Kapitel Strukturen zur Ablaufsteuerung
2. Kapitel Prozeduren: Unterprogramme und Funktionen
3. Kapitel Datei- und Geräte-E/A
4. Kapitel Zeichenkettenverarbeitung
5. Kapitel Graphiken
6. Kapitel Fehler- und Ereignisverfolgung
7. Kapitel Programmieren mit Modulen

2. Teil: BASIC-Grundbegriffe

8. Kapitel Zusammenfassung der Anweisungen und Funktionen
9. Kapitel Quick-Referenztabellen

Anhänge

Anhang A Übertragen von BASICA-Programmen in QuickBASIC
Anhang B Unterschiede zu früheren QuickBASIC-Versionen
Anhang C Einschränkungen für QuickBASIC
Anhang D Tastaturabfragecodes und ASCII-Zeichencodes
Anhang E Reservierte Wörter in BASIC
Anhang F Metabefehle
Anhang G Kompilieren und Binden aus DOS
Anhang H Erstellung und Verwendung von Quick-Bibliotheken
Anhang I Fehlermeldungen

Index

Inhaltsverzeichnis

Einleitung

Die QuickBASIC Programmiersprache xv

Die QuickBASIC Umgebung xvi

Verwendung dieses Handbuches xvii
Ausgewählte Programmierthemen xvii
BASIC-Grundbegriffe xviii
Anhänge xviii

Typographische Konventionen xix

Programmierstil dieses Handbuches xxi

1. Teil: Ausgewählte Programmierthemen

1 Strukturen zur Ablaufsteuerung

1.1 Die Reihenfolge der Ausführung von Anweisungen ändern 1.1

1.2 Boolesche Ausdrücke 1.2

1.3 Entscheidungsstrukturen 1.3
1.3.1 Block-IF...THEN...ELSE 1.8
1.3.2 SELECT CASE 1.10
1.3.2.1 Anwendung der SELECT CASE-Anweisung 1.12
1.3.2.2 SELECT CASE im Vergleich mit ON...GOSUB 1.16

1.4 Schleifenstrukturen 1.17
1.4.1 FOR...NEXT-Schleifen 1.17
1.4.1.1 Das Verlassen einer FOR...NEXT-Schleife anhand von EXIT FOR 1.22
1.4.1.2 Verzögerung der Programmausführung anhand einer FOR...NEXT-Schleife 1.22

1.4.2 WHILE...WEND-Schleifen 1.23
1.4.3 DO...LOOP-Schleifen 1.25
1.4.3.1 Schleifenprüfungen: eine Möglichkeit, DO...LOOP zu verlassen 1.29
1.4.3.2 EXIT DO: Eine Alternative, DO...LOOP zu verlassen 1.31

1.5 Anwendungsbeispiele 1.31
1.5.1 Programm zur Scheckheftsaldierung (*scheck.bas*) 1.32
1.5.2 Filter für Wagenrücklauf-Zeilenvorschub (*wrzv.bas*) 1.34

2 Prozeduren: Unterprogramme und Funktionen

2.1 Prozeduren: Bausteine für die Programmierung 2.2

2.2 Prozeduren im Vergleich mit Unterroutinen und Funktionen 2.2
2.2.1 SUB und GOSUB im Vergleich 2.3
2.2.1.1 Lokale und globale Variablen 2.3
2.2.1.2 Verwenden in Programmen mit mehreren Modulen 2.4
2.2.1.3 Arbeiten mit verschiedenen Sätzen von Variablen 2.4
2.2.2 FUNCTION im Vergleich mit DEF FN 2.5
2.2.2.1 Lokale und globale Variablen 2.5
2.2.2.2 Verändern von Variablen, die an Prozeduren übergeben werden 2.5
2.2.2.3 Aufruf der Prozedur aus ihrer eigenen Definition heraus 2.6
2.2.2.4 Verwendung in Programmen mit mehreren Modulen 2.6

2.3 Definieren der Prozeduren 2.7

2.4 Aufrufen von Prozeduren 2.9
2.4.1 Aufrufen einer FUNCTION-Prozedur 2.9
2.4.2 Aufrufen einer SUB-Prozedur 2.10

2.5 Übergabe von Argumenten an Prozeduren 2.11
2.5.1 Parameter und Argumente 2.12
2.5.2 Übergabe von Konstanten und Ausdrücken 2.14
2.5.3 Übergabe von Variablen 2.15
2.5.3.1 Übergabe von einfachen Variablen 2.15
2.5.3.2 Übergabe eines vollständigen Datenfeldes 2.16
2.5.3.3 Übergabe von einzelnen Datenfeldelementen 2.17
2.5.3.4 Verwendung der Funktionen für Datenfeldgrenzen 2.18
2.5.3.5 Übergabe eines vollständigen Datensatzes 2.18
2.5.3.6 Übergabe einzelner Elemente eines Datensatzes 2.19
2.5.4 Überprüfen von Argumenten anhand der DECLARE-Anweisung 2.20
2.5.4.1 Wann QuickBASIC keine DECLARE-Anweisung erzeugt 2.21
2.5.4.2 Entwicklung von Programmen außerhalb der QuickBASIC-Umgebung 2.21
2.5.4.3 Verwendung von Include-Dateien für Deklarationen 2.22
2.5.4.4 Deklarieren von Prozeduren in Quick-Bibliotheken 2.25
2.5.5 Übergabe von Argumenten als Referenz 2.26
2.5.6 Übergabe von Argumenten als Wert 2.27

2.6 Die gemeinsame Nutzung von Variablen anhand von SHARED 2.28
2.6.1 Die gemeinsame Nutzung von Variablen mit bestimmten Prozeduren in einem Modul 2.29
2.6.2 Die gemeinsame Nutzung von Variablen mit allen Prozeduren in einem Modul 2.30
2.6.3 Die gemeinsame Nutzung von Variablen mit anderen Modulen 2.33
2.6.4 Das Problem adreßgleicher Variablen 2.35
2.7 Automatische und STATIC-Variablen 2.36
2.8 Beibehaltung der Werte von lokalen Variablen mit Hilfe der STATIC-Anweisung 2.37
2.9 Rekursive Prozeduren 2.38
2.9.1 Die Funktion Fakultät 2.38
2.9.2 Anpassen der Stapelgröße 2.40
2.10 Übertragen der Kontrolle an ein anderes Programm anhand von CHAIN 2.40
2.11 Anwendungsbeispiel: Rekursive Verzeichnisdurchsuchung (*woist.bas*) 2.43

3 Datei- und Geräte-E/A

3.1 Ausgabe von Text auf den Bildschirm 3.2
3.1.1 Bildschirmzeilen- und spalten 3.2
3.1.2 Anzeige von Text und Zahlen anhand von PRINT 3.3
3.1.3 Anzeige von formatierten Ausgaben anhand von PRINT USING 3.5
3.1.4 Überspringen von Leerzeichen und Springen in eine angegebene Spalte 3.5
3.1.5 Ändern der Zeilen- und Spaltenzahl 3.6
3.1.6 Erstellen eines Text-Darstellungsfeldes 3.6
3.2 Einlesen der Eingabe von der Tastatur 3.8
3.2.1 Die INPUT-Anweisung 3.9
3.2.2 Die LINE INPUT-Anweisung 3.10
3.2.3 Die INPUT$-Funktion 3.12
3.2.4 Die INKEY$-Funktion 3.12
3.3 Steuerung des Text-Cursors 3.13
3.3.1 Positionieren des Cursors 3.14
3.3.2 Ändern der Cursor-Form 3.15
3.3.3 Informationen über die Position des Cursors 3.16
3.4 Arbeiten mit den Datendateien 3.17
3.4.1 Organisation der Datendateien 3.17
3.4.2 Sequentielle und Direktzugriffsdateien 3.18
3.4.3 Öffnen einer Datendatei 3.18
3.4.3.1 Dateinummern in BASIC 3.19
3.4.3.2 Dateinamen in BASIC 3.20

3.4.4 Schließen von Datendateien 3.21
3.4.5 Die Verwendung von sequentiellen Dateien 3.22
3.4.5.1 Datensätze in sequentiellen Dateien 3.22
3.4.5.2 Ergänzen einer neuen, sequentiellen Datei mit Daten 3.24
3.4.5.3 Lesen von Daten aus einer sequentiellen Datei 3.25
3.4.5.4 Ergänzen einer sequentiellen Datei mit Daten 3.26
3.4.5.5 Weitere Möglichkeiten, Daten in eine sequentielle Datei zu schreiben 3.26
3.4.5.6 Weitere Möglichkeiten, Daten aus einer sequentiellen Datei zu lesen 3.29
3.4.6 Die Verwendung von Direktzugriffsdateien 3.31
3.4.6.1 Datensätze in Direktzugriffsdateien 3.32
3.4.6.2 Ergänzen einer Direktzugriffsdatei mit Daten 3.32
3.4.6.3 Sequentielles Lesen von Daten 3.37
3.4.6.4 Zugriff auf bestimmte Direktzugriffsdateien anhand von Datensatznummern 3.39
3.4.7 Binärdatei-E/A 3.40
3.4.7.1 Vergleich zwischen Binärzugriff und Direktzugriff 3.41
3.4.7.2 Positionierung des Dateizeigers mit Hilfe von SEEK 3.41

3.5 Die Arbeit mit den Geräten 3.45
3.5.1 Unterschiede zwischen Geräte- und Datei-E/A 3.45
3.5.2 Datenübertragung über einen seriellen Anschluß 3.47

3.6 Anwendungsbeispiele 3.48
3.6.1 Immerwährender Kalender (*kal.bas*) 3.48
3.6.2 Indexieren einer Direktzugriffsdatei (*index.bas*) 3.54
3.6.3 Terminal-Emulator (*terminal.bas*) 3.62

4 Zeichenkettenverarbeitung

4.1 Definition der Zeichenketten 4.1

4.2 Zeichenketten variabler und fester Länge 4.3
4.2.1 Zeichenketten variabler Länge 4.3
4.2.2 Zeichenketten fester Länge 4.3

4.3 Kombinieren von Zeichenketten 4.5

4.4 Vergleichen von Zeichenketten 4.6

4.5 Suchen nach Zeichenketten 4.7

4.6 Ausgrenzen von Zeichenkettenteilen 4.9
4.6.1 Ausgrenzen der Zeichen von der linken Seite einer Zeichenkette 4.9
4.6.2 Ausgrenzen der Zeichen von der rechten Seite einer Zeichenkette 4.10
4.6.3 Ausgrenzen der Zeichen von einer beliebigen Stelle in einer Zeichenkette 4.11

4.7 Erzeugen von Zeichenketten 4.12

4.8 Verändern der Groß- bzw. Kleinschreibung 4.12

4.9 Zeichenketten und Zahlen 4.13

4.10 Verändern von Zeichenketten 4.14

4.11 Anwendungsbeispiel: Umsetzung einer Zeichenkette in eine Zahl (*zeiinzah.bas*) 4.15

5 Graphiken

5.1 Für graphische Programme erforderliche Voraussetzungen 5.2

5.2 Bildpunkte und Bildschirmkoordinaten 5.3

5.3 Das Zeichnen von Grundformen: Punkte, Geraden, Rechtecke und Kreise 5.4
- 5.3.1 Graphische Darstellung von Punkten anhand von PSET und PRESET 5.5
- 5.3.2 Zeichnen von Geraden und Rechtecken mit Hilfe von LINE 5.6
  - 5.3.2.1 Verwendung der Option STEP 5.6
  - 5.3.2.2 Zeichnen von Rechtecken 5.8
  - 5.3.2.3 Zeichnen von punktierten Geraden 5.10

5.4 Zeichnen von Kreisen und Ellipsen mit CIRCLE 5.11
- 5.4.1 Zeichnen von Kreisen 5.11
- 5.4.2 Zeichnen von Ellipsen 5.12
- 5.4.3 Zeichnen von Bögen 5.14
- 5.4.4 Zeichnen von Tortendiagrammen 5.17
- 5.4.5 Zeichnen von Formen, die mit dem Seitenverhältnis anzupassen sind 5.18

5.5 Definieren eines graphischen Darstellungsfeldes 5.20

5.6 Neu Definieren von Koordinaten eines Darstellungsfeldes anhand von WINDOW 5.24
- 5.6.1 Die Reihenfolge der Koordinatenpaare 5.28
- 5.6.2 Verfolgen der logischen und physikalischen Koordinaten 5.29

5.7 Die Verwendung von Farben 5.30
- 5.7.1 Wahl einer Farbe für graphische Ausgaben 5.31
- 5.7.2 Verändern der Vorder- oder Hintergrundfarbe 5.32
- 5.7.3 Verändern der Farben anhand von PALETTE und PALETTE USING 5.35

5.8 Ausmalen von Formen 5.37
- 5.8.1 Ausmalen mit Farben 5.38
- 5.8.2 Zeichnen mit Mustern: Ausfüllen mit Mustern 5.40
  - 5.8.2.1 Größe eines Kachelmusters in unterschiedlichen Bildschirmmodi 5.40
  - 5.8.2.2 Erzeugen eines einfarbigen Musters im Bildschirmmodus 2 5.41
  - 5.8.2.3 Erzeugen eines vielfarbigen Musters im Bildschirmmodus 1 5.44
  - 5.8.2.4 Erzeugen eines vielfarbigen Musters im Bildschirmmodus 8 5.48

5.9 DRAW: Eine graphische Makro-Sprache 5.50

5.10 Grundlegende Techniken der Animation 5.53
5.10.1 Speichern der Bilder mit GET 5.53
5.10.2 Bewegung der Bilder mit PUT 5.56
5.10.3 Animation mit Hilfe von GET und PUT 5.60
5.10.4 Animation mit Bildschirmseiten 5.66

5.11 Beispielanwendungen 5.68
5.11.1 Balkendiagramm-Generator (*balken.bas*) 5.68
5.11.2 Mathematisch erzeugte Farbe in einer Figur (*mandel.bas*) 5.74
5.11.3 Editor für Muster (*edmust.bas*) 5.81

6 Fehler- und Ereignisverfolgung

6.1 Fehlerverfolgung 6.1
6.1.1 Aktivierung der Fehlerverfolgung 6.2
6.1.2 Schreiben einer Fehlerbehandlungsroutine 6.2
6.1.2.1 Identifizieren von Fehlern mit Hilfe von ERR 6.3
6.1.2.2 Verlassen einer Fehlerbehandlungsroutine 6.5

6.2 Ereignisverfolgung 6.8
6.2.1 Erfassung der Ereignisse durch Registrierung 6.8
6.2.2 Erfassung der Ereignisse durch Verfolgung 6.8
6.2.3 Angabe des zu verfolgenden Ereignisses und Aktivierung der Ereignisverfolgung 6.9
6.2.4 Von BASIC verfolgte Ereignisse 6.10
6.2.5 Aussetzen oder Ausschalten der Ereignisverfolgung 6.10
6.2.6 Verfolgung der Tastenbetätigungen 6.12
6.2.6.1 Verfolgung der benutzerdefinierten Tasten 6.13
6.2.6.2 Verfolgung der benutzerdefinierten umgeschalteten Tasten 6.14
6.2.7 Verfolgung von Musikereignissen 6.17

6.3 Fehler- und Ereignisverfolgung in SUB- oder FUNCTION-Prozeduren 6.19

6.4 Verfolgung in Mehrfachmodulen 6.20
6.4.1 Ereignisverfolgung innerhalb von mehreren Modulen 6.20
6.4.2 Fehlerverfolgung innerhalb von mehreren Modulen 6.20

6.5 Fehler- und Ereignisverfolgung in Programmen, die mit BC kompiliert sind 6.25

6.6 Beispielanwendung: Verfolgen von Dateizugriffsfehlern (*datfehl.bas*) 6.26

7 Programmieren mit Modulen

7.1 Zweckmäßigkeit der Module 7.1

7.2 Hauptmodule 7.2

7.3 Module, die nur Prozeduren enthalten 7.2

7.4 Erstellen eines ausschließlich aus Prozeduren bestehenden Moduls 7.4

7.5 Laden der Module 7.4

7.6 Verwendung der DECLARE-Anweisung mit Mehrfachmodulen 7.5

7.7 Variablenzugriff von zwei oder mehreren Modulen 7.6
7.8 Verwendung der Module während der Programmentwicklung 7.7
7.9 Kompilieren und Binden von Modulen 7.7
7.10 Quick-Bibliotheken 7.8
7.10.1 Erstellen von Quick-Bibliotheken 7.9
7.11 Ratschläge für ein fachgerechtes Programmieren mit Modulen 7.10

2. Teil: BASIC-Grundbegriffe

8 Zusammenfassung der Anweisungen und Funktionen

9 Quick-Referenztabellen

9.1 Zusammenfassung der Anweisungen zur Ablaufsteuerung 9.2
9.2 Zusammenfassung der in BASIC-Prozeduren verwendeten Anweisungen 9.3
9.3 Zusammenfassung der Standard-E/A-Anweisungen 9.6
9.4 Zusammenfassung der Datei-E/A-Anweisungen 9.8
9.5 Zusammenfassung der Anweisungen und Funktionen zur Zeichenkettenverarbeitung 9.11
9.6 Zusammenfassung der Graphikanweisungen und -funktionen 9.14
9.7 Zusammenfassung der Anweisungen und Funktionen zur Fehler- und Ereignisverfolgung 9.16

Anhang A Übertragen von BASICA-Programmen in QuickBASIC

A.1 Format einer Quelldatei A.1
A.2 In QuickBASIC nicht zulässige Anweisungen und Funktionen A.2
A.3 Anweisungen, die Änderungen erfordern A.2
A.4 Editorunterschiede in der Handhabung von Tabulatoren A.3

Anhang B Unterschiede zu früheren QuickBASIC-Versionen

B.1 Eigenschaften von QuickBASIC B.2
B.1.1 Neue Eigenschaften von QuickBASIC 4.5 B.3
B.1.2 In QuickBASIC 4.0 eingeführte Eigenschaften B.4
B.1.2.1 Benutzerdefinierte Typen B.4
B.1.2.2 IEEE-Format und Unterstützung des mathematischen Koprozessors B.4
B.1.2.3 Bereiche der IEEE-Formatzahlen B.5
B.1.2.4 PRINT USING und Zahlen im IEEE-Format B.5
B.1.3 Neukompilieren alter Programme mit /MBF B.6

B.1.4 Konversion von Dateien und Programmen B.6
B.1.5 Andere QuickBASIC Eigenschaften B.8
B.1.5.1 Lange (32-Bit) Ganzzahlen B.8
B.1.5.2 Zeichenketten fester Länge B.8
B.1.5.3 Syntaxüberprüfung bei der Eingabe B.9
B.1.5.4 Binär-Datei-E/A B.9
B.1.5.5 FUNCTION-Prozeduren B.9
B.1.5.6 Unterstützung für den CodeView®-Debugger B.9
B.1.5.7 Kompatibilität zu anderen Sprachen B.10
B.1.5.8 Mehrere Module im Speicher B.10
B.1.5.9 Kompatibilität mit ProKey, SideKick und SuperKey B.10
B.1.5.10 Einfüge-/Überschreibemodus B.10
B.1.5.11 Tastaturbefehle im WordStar-Stil B.11
B.1.5.12 Rekursion B.11
B.1.5.13 Fehler-Listings während der getrennten Kompilierung B.11
B.1.5.14 Assembler-Listings während der getrennten Kompilierung B.11

B.2 Unterschiede in der Umgebung B.12
B.2.1 Befehls-und Optionsauswahl B.12
B.2.2 Fenster B.12
B.2.3 Das neue Menü B.12
B.2.4 Menübefehle B.12
B.2.5 Änderungen der Tastenkombinationen beim Bearbeiten B.14

B.3 Unterschiede beim Debuggen und Kompilieren B.15
B.3.1 Unterschiede in der Befehlszeile B.15
B.3.2 Unterschiede bei separater Kompilierung B.16
B.3.3 Benutzerbibliotheken und BUILDLIB B.17
B.3.4 Einschränkungen für Include-Dateien B.17
B.3.5 Debuggen B.18

B.4 Änderungen der Sprache BASIC B.19

B.5 Datei-Kompatibilität B.25

Anhang C Einschränkungen für QuickBASIC

Anhang D Tastaturabfragecodes und ASCII-Zeichencodes

D.1 Tastaturabfragecodes D.2

D.2 ASCII-Zeichencodes D.5

Anhang E Reservierte Wörter in QuickBASIC

Anhang F Metabefehle

F.1 Die Metabefehlssyntax F.1

F.2 Verarbeitung zusätzlicher Quelldateien: $INCLUDE F.2

F.3 Zuordnung von dimensionierten Datenfeldern: $STATIC und $DYNAMIC F.2

Anhang G Kompilieren und Binden aus DOS

G.1 BC, LINK und LIB G.2

G.2 Der Kompilier- und Bindeprozeß G.2

G.3 Kompilieren mit Hilfe des Befehls BC G.3

G.3.1 Angabe der Dateinamen G.4

G.3.1.1 Groß- und Kleinbuchstaben G.4

G.3.1.2 Erweiterungen für Dateinamen G.5

G.3.1.3 Pfadnamen G.5

G.3.2 Verwendung der BC-Befehlsoptionen G.5

G.4 Binden G.7

G.4.1 Vorgabewerte für LINK G.9

G.4.2 Dateiangabe für LINK G.11

G.4.3 Angabe von Bibliotheken für LINK G.11

G.4.4 Speicheranforderungen des Linkers G.12

G.4.5 Binden anhand von Programmen in verschiedenen Sprachen G.13

G.4.5.1 Pascal- und FORTRAN-Module in QuickBASIC-Programmen G.14

G.4.5.2 Zuordnung von STATIC-Datenfeldern in Routinen der Assembliersprache G.14

G.4.5.3 Bezugnahme auf DGROUP in erweiterten Laufzeitmodulen G.14

G.4.6 Verwendung der LINK-Optionen G.15

G.4.6.1 Anzeigen der Optionsliste (/HE) G.16

G.4.6.2 Anhalten während des Bindens (/PAU) G.16

G.4.6.3 Anzeigen von Informationen über den Bindevorgang(/I) G.17

G.4.6.4 Unterdrücken der Linker-Anfragen (/B) G.17

G.4.6.5 Erstellung von Quick-Bibliotheken (/Q) G.17

G.4.6.6 Packen der ausführbaren Dateien (/E) G.18

G.4.6.7 Ausschalten der Option Segment-Packen (/NOP) G.18

G.4.6.8 Ignorieren der üblichen BASIC-Bibliotheken (/NOD) G.18

G.4.6.9 Ignorieren der Wortverzeichnisse (/NOE) G.18

G.4.6.10 Setzen der Höchstanzahl an Segmenten (/SE) G.19

G.4.6.11 Erstellung einer Map-Datei (/M) G.19

G.4.6.12 Einfügen von Zeilennummern in eine Map-Datei (/LI) G.21

G.4.6.13 Packen angrenzender Segmente (/PAC) G.21

G.4.6.14 Der Einsatz des CodeView-Debuggers (/CO) G.22

G.4.6.15 Unterscheiden zwischen der Groß- und Kleinschreibung (/NOI) G.22

G.4.7 Andere LINK-Befehlszeilen-Optionen G.22

G.5 Verwaltung selbständiger Bibliotheken: LIB G.23

G.5.1 LIB starten G.24

G.5.2 Übliche Vorgaben für LIB G.26

G.5.3 List-Dateien mit Querverweisen G.27

G.5.4 Befehlssymbole G.27

G.5.5 LIB-Optionen G.29
G.5.5.1 Ignorieren der Groß- und Kleinschreibung bei Symbolen G.29
G.5.5.2 Ignorieren der erweiterten Wortverzeichnisse G.30
G.5.5.3 Unterscheiden der Symbolschreibweise G.30
G.5.5.4 Festlegen der Größe einer Bibliotheksseite G.30

Anhang H Erstellen und Verwenden von Quick-Bibliotheken

H.1 Bibliotheksarten H.1
H.2 Vorteile der Quick-Bibliotheken H.2
H.3 Anlegen einer Quick-Bibliothek H.3
H.3.1 Zum Anlegen einer Quick-Bibliothek erforderliche Dateien H.4
H.3.2 Aufbau einer Quick-Bibliothek H.4
H.3.3 Aufbau einer Quick-Bibliothek innerhalb der QuickBASIC-Umgebung H.5
H.3.3.1 Entfernen unerwünschter Dateien H.5
H.3.3.2 Laden gewünschter Dateien H.5
H.3.3.3 Erstellen einer Quick-Bibliothek H.6
H.4 Verwendung der Quick-Bibliotheken H.7
H.4.1 Laden einer Quick-Bibliothek H.7
H.4.2 Gleitkomma-Arithmetik in Quick-Bibliotheken H.8
H.4.3 Anzeigen des Inhalts einer Quick-Bibliothek H.8
H.5 Die mitgelieferte Bibliothek (*qb.qlb*) H.9
H.6 Die Dateinamenerweiterung *.qlb* H.9
H.7 Erstellen einer Bibliothek aus der Befehlszeile H.10
H.8 Verwendung anderssprachiger Routinen in einer Quick-Bibliothek H.10
H.8.1 Aufbau einer Quick-Bibliothek H.11
H.8.2 Quick-Bibliotheken mit führenden Nullen im ersten Code-Segment H.12
H.8.3 Die Routine B_OnExit H.12
H.9 Quick-Bibliotheken und Speicherplatz H.14
H.10 Erstellen kompakter ausführbarer Dateien H.14

Anhang I Fehlermeldungen

I.1 Anzeige der Fehlermeldungen I.2
I.2 Aufruf-, Kompilierzeit- und Laufzeitfehlermeldungen I.4
I.3 LINK-Fehlermeldungen I.34
I.4 LIB-Fehlermeldungen I.44

Index

Einleitung

Microsoft® QuickBASIC 4.5 leistet einen wichtigen Beitrag, BASIC's Leistungsfähigkeit zu erhöhen und die Benutzung gleichzeitig einfacher zu gestalten. Diese Version bietet die bisher fortschrittlichste, von einer Betriebskonfiguration unterstützte BASIC-Sprache für Microcomputer, die es dem Benutzer ermöglicht, sich auf die Programmentwicklung, nicht auf die Funktionsweise des Schreibvorganges oder die Fehlerbeseitigung, zu konzentrieren.

Die QuickBASIC Programmiersprache

Wenn Sie mit dem Programmieren in BASICA (oder einer ähnlichen interpretierten BASIC-Sprache) bereits vertraut sind, werden Sie die nachstehend aufgeführten verbesserten Spracheigenschaften von QuickBASIC, die das Schreiben und die Wartung der Software erleichtern, zu schätzen wissen:

- Die Anweisung **SELECT CASE** übergibt die Kontrolle an jeden Code-Block, ohne die verschachtelten Anweisungen **IF...THEN...ELSE**. **SELECT CASE** gewährleistet eine außergewöhnlich große Auswahl an Testausdrücken, mit deren Hilfe der am besten geeignete Vergleich angestellt werden kann.
- Die **SUB**- und **FUNCTION**-Prozeduren von QuickBASIC erlauben den Einsatz von Anweisungsgruppen des Programms in Unterprogramme, die vom Hauptprogramm wiederholt aufgerufen werden können. Dabei erleichtert die Modularität von QuickBASIC das Speichern dieser Prozeduren und deren Wiederverwendung in anderen Programmen.
- QuickBASIC-Prozeduren sind vollständig rekursiv, das bedeutet, daß sich eine Prozedur wiederholt aufrufen läßt. Diese Eigenschaft erleichtert die Programmierung zahlreicher numerischer und Sortieralgorithmen, die am besten rekursiv ausgedrückt werden können.

- Der Benutzer kann seine eigenen, aus einer beliebigen Kombination von Ganzzahlen, reellen und Zeichenkettenvariablen zusammengesetzten Datentypen definieren. Zusammenhängende Variablen können zweckmäßig unter einer einzigen Bezeichnung gruppiert werden, wodurch deren Übergabe an eine Prozedur oder deren Einsatz in eine Datei vereinfacht wird.
- QuickBASIC unterstützt den binären Dateizugriff. Die Programme können Dateien jedes Formats lesen und manipulieren, da die E/A über direkten Zugriff auf jedes Byte der Datei verfügt.

QuickBASIC stellt ein leistungsstarkes Entwicklungswerkzeug für den Berufsprogrammierer dar. Es bleibt jedoch auch die ideale Sprache für Programmieranfänger und Gelegenheitsprogrammierer, die zwar keine Berufsprogrammierer sind, aber eine Programmiersprache benötigen, die ihren Aufgaben und Zielen schnell und effizient gerecht wird.

Die QuickBASIC-Umgebung

QuickBASIC ist nicht nur eine hervorragende Programmiersprache, sie bildet auch eine integrierte Programmierumgebung, die das Schreiben der Software und die Fehlerbeseitigung erheblich erleichtert:

- Während der Programmeingabe prüft ein intelligenter Editor jede Zeile auf Syntaxfehler. Wenn Sie Ihr Programm starten wollen, brauchen Sie nur eine einzige Taste zu betätigen. Ein eventuell auftretendes Problem kann mit Hilfe des Editors gelöst und das Programm anschließend neu gestartet werden.
- Die Fehlerbeseitigung läßt sich innerhalb von QuickBASIC durchführen. Der integrierte Debugger erlaubt Ihnen, Variablen zu untersuchen und zu ändern, einen beliebigen Programmteil auszuführen oder die Ausführung beim Auftreten einer bestimmten Bedingung anzuhalten. Sämtliche Vorgänge finden in der QuickBASIC-Umgebung statt, ohne daß eine Änderung des Programms oder das Hinzufügen von **PRINT**-Anweisungen erforderlich ist.
- QuickBASIC 4.5 führt zwei neue Befehle ein, die zu einer gesteigerten Leistungsfähigkeit des Debuggers beitragen: den Befehl **Aktuellen Wert anzeigen** und den Befehl **Halt bei Fehler**.
- Der Microsoft QB-Ratgeber, ein angeschlossenes Hilfsprogramm, gehört ebenfalls zu den neuen Einrichtungen von QuickBASIC 4.5. Der QB-Ratgeber steht jederzeit, sowohl beim Schreiben, als auch während des Programmablaufes und der Fehlerbeseitigung, zu Ihrer Verfügung. Sie brauchen lediglich den Cursor auf das gewünschte Schlüsselwort oder die gewünschte benutzerdefinierte Bezeichnung zu positionieren und F1 zu drücken. Der QB-Ratgeber beschreibt die Syntax und Verwendung der BASIC-Anweisungen und -Funktionen und enthält anwendbare Programmierbeispiele.

Verwendung dieses Handbuches

Dieses Handbuch besteht aus drei Teilen. Der erste Teil, "Ausgewählte Programmierthemen", erteilt Informationen über bestimmte Programmiermethoden und -strategien. Der 2. Teil, "BASIC-Grundbegriffe", und die Anhänge enthalten wichtiges Referenzmaterial.

Ausgewählte Programmierthemen

Jedes Kapitel dieses ersten Abschnittes erläutert einen der nachstehenden Programmierbereiche und hilft dem Benutzer, diese schnell zu beherrschen:

- Strukturen zur Ablaufsteuerung
- **SUB**- und **FUNCTION**-Prozeduren
- Datei- und Geräte-E/A
- Zeichenkettenverarbeitung
- Graphik
- Fehler- und Ereignisverfolgung
- Programmieren mit Modulen

Diese Themen werden einfach und verständlich vorgestellt und anhand zahlreicher kurzer Programmierbeispiele veranschaulicht, die die Arbeitsweise der BASIC-Bestandteile darstellen. Die Erklärungen führen von einfachen zu komplexeren Themen, so daß Sie das Material durchgehen können, ohne auf die Reihenfolge der Themen zu achten. Dieses Handbuch konzentriert sich in erster Reihe auf die Anwendung, nicht die Theorie, zur Lösung häufig vorkommender Probleme anhand von QuickBASIC.

Zusätzlich zu den kurzen Beispielen enthalten die Kapitel vollständige funktionierende Programme, die die in dem jeweiligen Kapitel vorgestellten Programmiergrundsätze in Praxis umsetzen. Diese Programme werden in dem QuickBASIC-Paket mitgeliefert.

Als erfahrener Programmierer werden Sie wahrscheinlich das Inhaltsverzeichnis auf der Suche nach einem interessanten Thema durchblättern. Für einen Programmieranfänger empfiehlt es sich jedoch, die Kapitel nacheinander von Anfang bis Ende durchzugehen. Wenn Ihnen keine Programmiersprache bekannt ist, sollten Sie mit Kapitel 4, "BASIC für Anfänger", des Handbuches *Lernen und Anwenden von Microsoft QuickBASIC* beginnen.

Diese sieben Kapitel werden Ihnen helfen, unabhängig von Ihren Interessen und Erfahrungen, die für das Schreiben von komplexen BASIC-Anwendungsprogrammen erforderlichen Kenntnisse zu erlernen.

BASIC-Grundbegriffe

Der zweite Teil dieses Handbuchs, "BASIC-Grundbegriffe", besteht aus zwei Teilen, die sich auf BASIC-Anweisungen und Funktionen beziehen.

Kapitel 8, "Zusammenfassung der Anweisungen und Funktionen", stellt eine alphabetische Zusammenfassung aller BASIC-Schlüsselworte dar, in der auch deren Funktion, Anwendung und Syntax beschrieben werden. Dieser Abschnitt soll als Gedächtnisstütze bei der Verwendung von Anweisungen und Funktionen dienen.

Kapitel 9, "Quick-Referenztabellen", stellt die am häufigsten verwendeten BASIC-Anweisungen und -Funktionen mit der entsprechenden Kurzbeschreibung tabellenförmig in sieben Abschnitten vor. Der Inhalt dieser Abschnitte stimmt mit dem in den Kapiteln 1 bis 6 der "Ausgewählten Programmierthemen" vorgestellten Material überein. Dieses Kapitel wird Ihnen bei der Lösung bestimmter Programmieraufgaben behilflich sein.

Anhänge

Der dritte Abschnitt dieses Handbuches besteht aus mehreren Anhängen, die sich auf folgende Bereiche beziehen:

- Übertragen von BASICA-Programmen
- Unterschiede zu vorhergehenden Versionen
- Einschränkungen für QuickBASIC
- Tastaturabfragecodes und ASCII-Codes
- Reservierte Wörter für BASIC
- Metabefehle
- Kompilieren und Binden aus DOS
- Erstellung und Verwendung von Quick-Bibliotheken
- Fehlermeldungen

Typographische Konventionen

In diesem Handbuch werden folgende typographische Konventionen verwendet:

Darstellung	***Beschreibung***
COPY, LINK, /X	Großbuchstaben zeigen Befehle auf DOS-Ebene an. Großbuchstaben werden auch für Befehlszeilen-Optionen benutzt, es sei denn, die Anwendung nimmt nur Kleinbuchstaben an.
SUB, IF, LOOP, PRINT, WHILE, TIME$	Fettgedruckte Großbuchstaben zeigen sprachspezifische Schlüsselwörter mit besonderer Bedeutung in Microsoft BASIC an. Diese Schlüsselwörter sind ein notwendiger Bestandteil der Anweisungs-Syntax, es sei denn, sie sind, wie nachfolgend beschrieben, in Klammern eingeschlossen. Beim Schreiben von Programmen muß die Schlüsselworteingabe genau wie gezeigt erfolgen. Es können jedoch dabei sowohl Groß- als auch Kleinbuchstaben verwendet werden.
qb.lib, add.exe	Kursive Kleinbuchstaben geben Dateinamen oder -erweiterungen an.
`CALLNewProc (Arg1!, Var2%)`	Diese Schriftart wird zur Darstellung von Programmbeispielen, Programmausgabe und Fehlermeldungen innerhalb des Textes benutzt.

`'$INCLUDE:'BC.BI'`
`.`
`.`
`.`
`CHAIN "PROG1"`
`END`
	Eine dreipunktige Spalte zeigt an, daß ein Programmteil absichtlich ausgelassen wurde.

`' Führe eine Übergabe` `' durch`	Der Apostroph (rechtes einzelnes Anführungszeichen) kennzeichnet den Anfang eines Kommentars in Beispielprogrammen.

Darstellung	***Beschreibung***
Dateiangabe	Begriffe in Kursivschrift sind Platzhalter für einzugebende Informationen, wie Dateinamen. Die Kursivschrift wird gelegentlich auch zum Unterstreichen einer Textpassage benutzt.
[*Wahlweiser Begriff*]	Zwischen eckigen Klammern stehende Begriffe sind optional.
{*Wahl1* | *Wahl2*}	Geschweifte Klammern und vertikale Balken zeigen eine Auswahl zwischen zwei oder mehreren Begriffen an. Der Benutzer muß eine Wahl treffen, es sei denn, daß sämtliche Begriffe auch in eckigen Klammern stehen.
Wiederholende Elemente...	Drei nach einem Begriff stehende Punkte deuten darauf hin, daß mehrere Begriffe derselben Form auftreten können.
Unterstrich (_)	Der Unterstrich kennzeichnet in Beispielprogrammen, daß die BASICzeile hier nicht endet, sondern in der nächsten Zeile im Handbuch fortgesetzt wird.
ALT+F1	Kleingedruckte Großbuchstaben werden für die Bezeichnungen von Tasten und Tastenkombinationen wie EINGABETASTE und STRG+R benutzt.
	Ein Pluszeichen (+) weist auf eine Tastenkombination hin. Zum Beispiel weist STRG+E den Benutzer an, die E-TASTE bei niedergedrückter STRG-TASTE zu betätigen.
	Die manchmal mit einem gekrümmten Pfeil gekennzeichnete Wagenrücklauftaste wird als EINGABETASTE bezeichnet.
	Die im rechten Tastenfeld vorhandenen Cursor-Bewegungstasten ("Pfeiltasten") werden RICHTUNGSTASTEN genannt. Einzelne RICHTUNGSTASTEN werden entweder durch die Richtung des Pfeiles auf der Taste bezeichnet (NACH LINKS, NACH RECHTS, NACH OBEN, NACH UNTEN) oder durch die Bezeichnung auf der Taste (BILD↑, BILD↓).

Darstellung	***Beschreibung***
	Die in diesem Handbuch verwendeten Tastenbezeichnungen entsprechen den auf den Tasten des IBM-Personal Computers stehenden Bezeichnungen. Andere Computer können abweichende Bezeichnungen benutzen.
"Definierter Begriff"	Anführungsstriche kennzeichnen normalerweise einen neuen im Text definierten Begriff.
Video Graphics Array (VGA)	Abkürzungen werden üblicherweise ausgeschrieben, wenn sie zum ersten Mal im Text auftreten.

Die untenstehende Syntax (der Anweisungen "**LOCK...UNLOCK**") veranschaulicht viele der typographischen Konventionen dieses Handbuches:

LOCK [#] *Dateinummer* [,{*Datensatz* | [*Beginn*] **TO** *Ende*}]

.

.

.

UNLOCK [#] *Dateinummer* [,{*Datensatz*| [*Beginn*] **TO** *Ende*}]

Hinweis In diesem Handbuch bezeichnet der Begriff "DOS" sowohl MS-DOS® als auch das Betriebssystem IBM Personal Computer DOS. Der Name eines bestimmten Betriebssystems wird bei dem Hinweis auf spezifische Eigenschaften des betreffenden Systems verwendet. Der Begriff "BASICA" bezieht sich auf alle interpretierten BASIC-Versionen im allgemeinen.

Programmierstil dieses Handbuches

Es folgen die Richtlinien, die beim Schreiben der in diesem Handbuch und auf den Originaldisketten vorhandenen Programme angewandt wurden. Obwohl diese Richtlinien für die Programmlesbarkeit empfohlen werden, müssen Sie sich beim Schreiben eigener Programme nicht danach richten.

- Schlüsselwörter und symbolische Konstanten werden in Großbuchstaben dargestellt:

```
' PRINT, DO, LOOP, UNTIL sind Schlüsselwörter:
PRINT "Titelseite"
DO LOOP UNTIL Antwort$ = "N"

' FALSCH und WAHR sind symbolische Konstanten
' gleich 0 bzw. -1:
CONST FALSCH = 0, WAHR = NOT FALSCH
```

- Variablennamen werden in Kleinbuchstaben mit einem großen Anfangsbuchstaben dargestellt; Variablennamen mit mehr als einer Silbe können weitere Großbuchstaben enthalten, um den Silbenanfang zu kennzeichnen:

```
AnzSaetze% = 45
GebDat$ = "28.10.63"
```

- Zeilenmarken werden anstelle von Zeilennummern verwendet. Zeilenmarken werden lediglich bei Routinen zur Ereignisverfolgung und Fehlerbehandlung benutzt, sowie bei **DATA**-Anweisungen, auf die mit **RESTORE** Bezug genommen wird.

```
' Zeitbehand und BildZweiDaten sind Zeilenmarken:
ON TIMER GOSUB ZeitBehand
RESTORE BildZweiDaten
```

- Wie bereits im vorhergehenden Abschnitt erwähnt, leitet ein einzelnes Apostroph (') Kommentare ein:

```
' Dies ist ein Kommentar; diese beiden Zeilen werden vom
' Programm während der Ausführung ignoriert.
```

- Blöcke zur Ablaufsteuerung und Anweisungen in Prozeduren oder Unterroutinen werden wie folgt innerhalb des umschließenden Codes eingerückt:

```
SUB Eingabe STATIC
   FOR I% = 1 TO 10
      INPUT X
      IF X > 0 THEN
         .
         .
         .
      ELSE
         .
         .
         .
      END IF
   NEXT I%
END SUB
```

1.Teil: Ausgewählte Programmierthemen

1. Teil

```
AnzDaten% = AnzDaten% + 1
PRINT
PRINT "Balken ("; LTRIM$ (STR$ (AnzDaten% )) ; ") : "
INPUT ;  "Bezeichnung? ", Bezeich$ (AnzDaten%)
' Eingabe eines Wertes nur, wenn Bezeichnung
' nicht leer ist:
IF Bezeich$ (AnzDaten%) <> "" THEN
    entf = LEN (Bezeich$ (AnzDaten%)) + 30
    LOCATE ,  entf
    INPUT "Wert ? ", Wert (AnzDaten%)
' Wenn Bezeichnung leer ist, vermindere den
' Datenzähler und setze Flag für Fertig auf
' WAHR:
ELSE
```

Ausgewählte Programmierthemen

Der 1. Teil führt die Grundlage der BASIC-Programmierung ein, wobei zuerst einfache Themen behandelt werden.

Das 1. Kapitel erläutert die Strukturen zur Ablaufsteuerung, die die Programmausführung steuern. Das 2. Kapitel beschreibt QuickBASIC Unterprogramme und Funktionen, die zwei sehr leistungsfähige Programmierhilfen darstellen. Kapitel 3 veranschaulicht die Anwendung von QuickBASIC bei der Arbeit mit den vom Programm angenommenen und erzeugten Daten. Kapitel 4 behandelt die Verwendung von Textzeichenketten und Kapitel 5 stellt die graphischen Fähigkeiten von QuickBASIC vor.

Die letzten beiden Kapitel umfassen Themen für Fortgeschrittene. Kapitel 6 erklärt die Fehler- und Ereignisverfolgung und Kapitel 7 zeigt die Vorteile des Programmierens mit Modulen.

Kapitelverzeichnis

1 Strukturen zur Ablaufsteuerung
2 Prozeduren: Unterprogramme und Funktionen
3 Datei- und Geräte-E/A
4 Zeichenkettenverarbeitung
5 Graphiken
6 Fehler- und Ereignisverfolgung
7 Programmieren mit Modulen

1 Strukturen zur Ablaufsteuerung

Dieses Kapitel zeigt Ihnen, wie Sie Strukturen zur Ablaufsteuerung verwenden, insbesondere Schleifen und Entscheidungsanweisungen, um den Ablauf der Ausführung Ihres Programmes zu steuern. Schleifen veranlassen ein Programm, eine Reihe von Anweisungen beliebig oft zu wiederholen. Entscheidungsanweisungen lassen das Programm entscheiden, welcher der verschiedenen möglichen Pfade auszuwählen ist.

Am Ende dieses Kapitels werden Sie in der Lage sein, folgende Aufgaben im Zusammenhang mit dem Einsatz von Schleifen und Entscheidungsanweisungen in Ihren BASIC-Programmen durchzuführen:

- Vergleichen von Ausdrücken anhand von Vergleichsoperatoren.
- Kombinieren von numerischen oder Stringausdrücken mit logischen Operatoren und Feststellen, ob der resultierende Ausdruck wahr oder falsch ist.
- Erstellen von Verzweigungen im Programmablauf aufgrund der Anweisungen **IF...THEN...ELSE** und **SELECT CASE.**
- Schreiben von Schleifen, die Anweisungen sooft wie angegeben wiederholen.
- Schreiben von Schleifen, die Anweisungen wiederholen, solange oder bis eine bestimmte Bedingung wahr ist.

1.1 Die Reihenfolge der Ausführung von Anweisungen ändern

Wenn die Logik eines Programmes nicht von Anweisungen zur Ablaufsteuerung kontrolliert wird, durchläuft sie die Anweisungen von links nach rechts und von oben nach unten. Während einige sehr einfache Programme nur mit diesem direkten Ablauf geschrieben werden können, wird die Stärke und Verwendbarkeit einer Programmiersprache größtenteils von deren Fähigkeit bestimmt, die Reihenfolge der Ausführung von Anweisungen durch Entscheidungsstrukturen und Schleifen zu ändern.

Das Programm kann anhand einer Entscheidungsstruktur einen Ausdruck auswerten und anschließend, je nach dem Ergebnis der Auswertung, in einen der verschiedenen Anweisungsblöcke verzweigen. Mit einer Schleife kann ein Programm Anweisungsblöcke wiederholt ausführen.

Falls Sie bereits in BASICA programmiert haben, werden Sie die größere Flexibilität folgender zusätzlichen Strukturen zur Ablaufsteuerung in dieser BASIC-Version zu schätzen wissen:

- Die Block-**IF...THEN...ELSE**-Anweisung.
- Die **SELECT CASE**-Anweisung.
- Die **DO...LOOP**- und **EXIT DO**-Anweisungen.
- Die **EXIT FOR**-Anweisung, die eine Alternative zum Verlassen von **FOR...NEXT**-Schleifen bietet.

1.2 Boolesche Ausdrücke

Ein Boolescher Ausdruck ist jeder Ausdruck, der den Wert "wahr" oder "falsch" liefert. BASIC verwendet Boolesche Ausdrücke in bestimmten Arten von Entscheidungsstrukturen und Schleifen. Nachstehende **IF...THEN...ELSE**-Anweisung enthält einen Booleschen Ausdruck, `X < Y`:

```
IF X < Y THEN CALL Prozedur1 ELSE CALL Prozedur2
```

Wenn in diesem Beispiel der Boolesche Ausdruck wahr ist (dies bedeutet, daß der Wert der Variablen `X` tatsächlich kleiner als der Wert der Variablen `Y` ist), wird `Prozedur1` ausgeführt; anderenfalls (wenn `X` größer oder gleich `Y` ist) wird `Prozedur2` ausgeführt.

Das vorhergehende Beispiel veranschaulicht auch die übliche Anwendung Boolescher Ausdrücke: zwei verschiedene Ausdrücke (in diesem Fall `X` und `Y`) werden verglichen, um deren Beziehung zueinander zu bestimmen. Diese Vergleiche werden anhand der in Tabelle 1.1 dargestellten Vergleichsoperatoren durchgeführt:

Tabelle 1.1 In BASIC verwendete Vergleichsoperatoren

Operator	***Bedeutung***
=	Gleich
<>	Ungleich
<	Kleiner als
<=	Kleiner gleich
>	Größer als
>=	Größer gleich

Diese Vergleichsoperatoren können zum Vergleichen von Zeichenkettenausdrücken verwendet werden. In diesem Fall beziehen sich "größer", "kleiner", u.s.w. auf die alphabetische Reihenfolge. Der folgende Ausdruck, zum Beispiel, ist wahr, da das Wort "andere" in der alphabetischen Reihenfolge vor dem Wort "anders" steht:

```
"andere" < "anders"
```

Boolesche Ausdrücke verwenden ebenfalls häufig die "logischen Operatoren" **AND**, **OR**, **NOT**, **XOR**, **IMP** und **EQV**. Diese Operatoren ermöglichen den Aufbau zusammengesetzter Tests aufgrund eines oder mehrerer Boolescher Ausdrücke. Zum Beispiel ist:

Ausdruck1 **AND** *Ausdruck2*

nur dann wahr, wenn sowohl *Ausdruck1* als auch *Ausdruck2* wahr sind. Demnach wird die Meldung `Alles sortiert` im folgenden Beispiel nur dann ausgegeben, wenn beide Booleschen Ausdrücke `X <= Y` und `Y <= Z` wahr sind:

```
IF (X <= Y) AND (Y <= Z) THEN PRINT "Alles sortiert"
```

Im vorhergehenden Beispiel sind die Klammern um den Booleschen Ausdruck nicht unbedingt erforderlich, da Vergleichsoperatoren wie <= vor logischen Operatoren wie **AND** ausgewertet werden. Klammern tragen jedoch zu der lesbaren Gestaltung eines komplexen Booleschen Ausdruckes bei und gewährleisten darüber hinaus die Auswertung seiner Bestandteile in der gewünschten Reihenfolge.

BASIC verwendet die numerischen Werte -1 und 0, um wahr bzw. falsch darzustellen. Dies wird im nächsten Beispiel deutlich, in dem die Ausgabe eines wahren und eines falschen Ausdruckes veranlaßt wird:

```
x = 5
y = 10
PRINT x < y   ' Auswertung, Ausgabe eines "wahren" Booleschen
              ' Ausdruckes.
PRINT x < y   ' Auswertung, Ausgabe eines "falschen"
              ' Booleschen Ausdruckes.
```

Ausgabe

```
-1
 0
```

Der Wert -1 für wahr wird einleuchtender, wenn man die Arbeitsweise des **NOT**-Operators von BASIC betrachtet: **NOT** invertiert jedes Bit der binären Darstellung seines Operanden durch Ändern von 1-Bits in 0-Bits und von 0-Bits in 1-Bits. Da der ganzzahlige Wert 0 (falsch) als eine Folge von sechzehn 0-Bits intern gespeichert ist, wird **NOT** 0 (wahr) intern als sechzehn 1-Bits wie folgt gespeichert:

```
FALSCH             = 0000000000000000
WAHR = NOT FALSCH  = 1111111111111111
```

In der Zweierkomplementmethode, die BASIC zum Speichern von Ganzzahlen verwendet, stellen sechzehn 1-Bits den Wert -1 dar.

Es ist zu beachten, daß BASIC -1 selbst ausgibt, wenn es einen Booleschen Ausdruck als wahr auswertet. Wie die Ausgabe im folgenden Beispiel zeigt, betrachtet BASIC jedoch jeden Wert ungleich Null als wahr:

```
INPUT "Geben Sie einen Wert ein: ", x
IF x THEN PRINT x "ist wahr."
```

Ausgabe

```
Geben Sie einen Wert ein: 2
2 ist wahr.
```

In BASIC ist der **NOT**-Operator als "bitweiser" Operator bekannt. Einige Programmiersprachen, wie C oder Pascal, verfügen sowohl über einen bitweisen **NOT**-Operator als auch über einen "logischen" **NOT**-Operator, die sich durch folgendes unterscheiden:

- Ein bitweiser **NOT** gibt falsch (0) nur für den Wert -1 aus.
- Ein logischer **NOT** gibt falsch (0) für jeden wahren (ungleich Null) Wert aus.

Wie in nachstehender Liste aufgeführt, gibt in BASIC der **NOT** *Ausdruck* für jeden wahren *Ausdruck* ungleich -1 einen anderen wahren Wert aus:

Wert des Ausdruckes	***Wert des NOT-Ausdruckes***
1	-2
2	-3
-2	1
-1	0

Folglich ist zu beachten: **NOT** *Ausdruck* ist nur dann falsch, wenn *Ausdruck* einen Wert von -1 ergibt. Werden Boolesche Konstanten oder Variablen für die Verwendung im Programm definiert, ist -1 für wahr einzusetzen.

Die Werte 0 und -1 können bei der Definition von hilfreichen mnemonischen Booleschen Konstanten benutzt werden, die in Schleifen und Entscheidungen zur Anwendung kommen. Wie im folgenden Programmausschnitt gezeigt, der die Elemente eines als `Betrag` bezeichneten Datenfeldes in aufsteigender Reihenfolge sortiert, wird diese Technik häufig in den Beispielen dieses Handbuchs verwendet:

```
' Definiere symbolische Konstanten zur Verwendung im
' Programm:
CONST FALSCH = 0, WAHR = NOT FALSCH
.
.
.
DO
   Tausch = FALSCH
   FOR I = 1 TO TransakAnz-1
      IF Betrag(I) < Betrag(I+1) THEN
         SWAP Betrag(I), Betrag(I+1)
         Tausch = WAHR
      END IF
   NEXT I
LOOP WHILE Tausch        ' Durchlaufe die Schleife solange
                         ' Tausch WAHR ist.
.
.
.
```

1.3 Entscheidungsstrukturen

Ausgehend vom Wert eines Ausdruckes veranlassen Entscheidungsstrukturen ein Programm eine der folgenden Aktionen durchzuführen:

1. Ausführen einer der verschiedenen, innerhalb der Entscheidungsstruktur vorhandenen alternativen Anweisungen.
2. Verzweigen in einen anderen Teil des Programms außerhalb der Entscheidungsstruktur.

In BASICA werden Entscheidungen nur mit der einzeiligen **IF...THEN [...ELSE]**-Anweisung bearbeitet. In ihrer einfachsten Form (**IF...THEN**) wird der Ausdruck ausgewertet, der dem Schlüsselwort **IF** folgt. Ist dieser Ausdruck wahr, führt das Programm die Anweisungen aus, die dem Schlüsselwort **THEN** folgen; ist der Ausdruck falsch, fährt das Programm mit der nächsten Zeile nach der **IF...THEN**-Anweisung fort. Die Zeilen 50 und 70 des folgenden BASICA-Programmausschnittes zeigen Beispiele für **IF...THEN**:

```
30  INPUT A
40  ' Wenn A größer als 100 ist, gib eine Meldung aus und
45  ' gehe zurück zu Zeile 30; andernfalls gehe zu Zeile 60:
50  ' IF A > 100 THEN PRINT "Zu groß": GOTO 30
60  ' Wenn A gleich 100 ist, springe zu Zeile 300;
65  ' andernfalls, gehe zu Zeile 80:
70  IF A = 100 THEN GOTO 300
80  PRINT A/100: GOTO 30
.
.
.
```

Durch Ergänzen einer **IF...THEN**-Anweisung mit einer **ELSE**-Klausel kann ein Programm veranlaßt werden, eine Reihe von Aktionen (die dem Schlüsselwort **THEN** folgen) im Fall eines wahren Ausdruckes und eine andere Reihe von Aktionen (die dem Schlüsselwort **THEN** folgen) im Fall eines falschen Ausdruckes auszuführen. Der nächste Programmausschnitt veranschaulicht, wie **ELSE** in einer **IF...THEN...ELSE**-Anweisung funktioniert:

```
10  INPUT "Geben Sie Ihre Kennung ein"; Kenn$
15  ' Wenn der Benutzer "Andrea" eingibt, gehe zu Zeile 50;
20  ' andernfalls gib eine Meldung aus und gehe zurück zu
25  ' Zeile 10:
30  IF Kenn$="Andrea" THEN 50 ELSE PRINT "Versuchen Sie_
    es noch einmal": GOTO 10
.
.
.
```

Während das einzeilige **IF...THEN...ELSE** von BASICA für einfache Entscheidungen zweckmäßig ist, führt es bei komplizierteren Entscheidungen zu wirklich unlesbaren Codes. Dies ist besonders dann der Fall, wenn die Programme so gestaltet sind, daß alle alternativen Aktionen innerhalb der **IF...THEN...ELSE**-Anweisung selbst stattfinden, oder wenn **IF...THEN...ELSE**-Anweisungen verschachtelt werden (das heißt, wenn eine **IF...THEN...ELSE** innerhalb einer anderen plaziert wird, eine durchaus zulässige Konstruktion). Anhand des folgenden Ausschnitts aus einem BASICA-Programm können Sie feststellen, wie schwierig es ist, selbst einen einfachen Test unter diesen Bedingungen zu verfolgen:

```
10  ' Die folgenden verschachtelten IF...THEN...ELSE
15  ' Anweisungen schreiben verschiedene Ausgaben
20  ' für jeden der vier folgenden Fälle:
25  '    1) A <= 50, B <= 50     3) A > 50, B <= 50
30  '    2) A <= 50, B > 50      4) A > 50, B > 50
35
40  INPUT A, B
45
50  ' Beachte: Obwohl Zeile 70 über mehrere physikalische
55  ' Zeilen des Bildschirms geht, ist sie nur eine
60  ' "logische Zeile"
65  ' (alles, was vor Betätigung der Eingabetaste eingetippt
70  ' wurde) BASICA bricht lange Zeilen auf dem Bildschirm
75  ' um.
80  IF A <= 50 THEN IF B <= 50 THEN PRINT "A <= 50, B <= 50"
    ELSE PRINT   "A <= 50, B > 50" ELSE IF B <= 50 THEN
    PRINT "A > 50, B <= 50" ELSE PRINT "A > 50, B > 50"
```

Um diese Art komplizierter Anweisungen zu vermeiden, bietet BASIC nun die **IF...THEN...ELSE**-Anweisung in Blockform, sodaß eine Entscheidung nicht länger auf eine logische Zeile begrenzt ist. Nachstehend wird dasselbe, unter Verwendung von Block-**IF...THEN...ELSE** umgeschriebene BASICA-Programm dargestellt:

```
INPUT A, B
IF A <= 50 THEN
   IF B <= 50 THEN
      PRINT "A <= 50, B <= 50"
   ELSE
      PRINT "A <= 50, B > 50"
   END IF
ELSE
   IF B <= 50 THEN
      PRINT "A > 50, B <= 50"
   ELSE
      PRINT "A > 50, B > 50"
   END IF
END IF
```

Für strukturierte Entscheidungen kennt QuickBASIC ebenso die Anweisung **SELECT CASE...END SELECT** (im folgenden als **SELECT CASE** bezeichnet).

Sowohl die Block-**IF...THEN...ELSE**-Anweisung als auch die **SELECT CASE**-Anweisung erlauben die Gestaltung des Programmcodes nach logischen Gesichtspunkten, nicht nach der Anzahl der in einer Zeile untergebrachten Anweisungen. Diese Anweisungen gewährleisten erhöhte Flexibilität während der Programmierung, sowie verbesserte Programmlesbarkeit und leichte Wartung.

1.3.1 Block-IF...THEN...ELSE

Tabelle 1.2 veranschaulicht die Syntax der **IF...THEN...ELSE**-Anweisung und deren Anwendung:

Tabelle 1.2 Syntax und Beispiel einer Block-IF...THEN...ELSE-Anweisung

Syntax

IF *Bedingung1* **THEN**
  [*Anweisungsblock-1*]
[**ELSEIF** *Bedingung2* **THEN**
  [*Anweisungsblock-2*]]
.
.
.
[**ELSE**
  [*Anweisungsblock-n*]]
END IF

Beispiel

```
IF X > 0 THEN
 PRINT "X ist positiv"
 PosZahl = PosZahl + 1
ELSEIF X < 0 THEN
 PRINT "X ist negativ"
 NegZahl = NegZahl + 1
ELSE
 PRINT "X ist Null"
END IF
```

Die Argumente *Bedingung1*, *Bedingung2*, usw. sind Ausdrücke. Sie können sowohl numerische Ausdrücke - in diesem Fall ist jeder Wert ungleich Null wahr und Null ist falsch - als auch Boolesche Ausdrücke sein, wobei -1 wahr und Null falsch ist. Wie in Abschnitt 1.2 erläutert, vergleichen Boolesche Ausdrücke typischerweise zwei numerische oder Zeichenkettenausdrücke anhand von Vergleichsoperatoren wie < oder >=.

Nach jeder **IF**-, **ELSEIF**- und **ELSE**-Klausel steht ein Anweisungsblock. Keine der Anweisungen darf sich auf der gleichen Zeile wie die **IF**-, **ELSEIF**- oder **ELSE**-Klausel befinden, um nicht von BASIC als einzeilige **IF...THEN**-Anweisung aufgefaßt zu werden.

BASIC wertet jegliche Ausdrücke der **IF**- und **ELSEIF**-Klauseln von oben nach unten aus und übergeht Anweisungsblöcke, bis es den ersten wahren Ausdruck findet. Wenn es einen wahren Ausdruck findet, führt es die zu dem Ausdruck gehörenden Anweisungen aus und verzweigt dann aus dem Block heraus zu der Anweisung, die der **END IF**-Klausel folgt.

Ist keiner der Ausdrücke in der **IF**- oder **ELSEIF**-Klausel wahr, springt BASIC gegebenenfalls zu der **ELSE**-Klausel und führt deren Anweisungen aus. Ist jedoch keine **ELSE**-Klausel vorhanden, fährt das Programm mit der nächsten Anweisung nach der **END IF**-Klausel fort.

Wie im nachfolgenden Beispiel gezeigt, sind die **ELSE**- und **ELSEIF**-Klauseln beide optional:

```
' Wenn der Wert von X kleiner als 100 ist, führe beide
' Anweisungen vor END IF aus;
' andernfalls gehe zu der INPUT-Anweisung nach END IF:
' IF X < 100 THEN
   PRINT X
   Nummer = Nummer + 1
END IF
INPUT "Neuer Wert"; Antwort$
.
.
.
```

Eine einzelne **IF...THEN...ELSE** kann, wie unten dargestellt, mehrere **ELSEIF**-Anweisungen enthalten:

```
IF C$ >= "A" AND C$ <= "Z" THEN
   PRINT "Großbuchstaben"
ELSEIF C$ >= "a" AND C$ <= "z" THEN
   PRINT "Kleinbuchstaben"
ELSEIF C$ >= "0" AND C$ <= "9" THEN
   PRINT "Zahl"
ELSE
   PRINT "Nicht alphanumerisch"
END IF
```

Es wird höchstens ein Block von Anweisungen ausgeführt, selbst wenn mehr als eine Bedingung wahr ist. Wenn Sie beispielsweise das Wort `als` in die Eingabe des nächsten Beispiels schreiben, erscheint die Meldung `Eingabe zu kurz`, jedoch nicht die Meldung `Kann nicht mit a beginnen`:

```
INPUT Checks$
IF LEN(Check$) > 6 THEN
   PRINT "Eingabe zu lang"
ELSEIF LEN(Check$) < 6 THEN
   PRINT "Eingabe zu kurz"
ELSEIF LEFT$(Check$, 1) = "a" THEN
   PRINT "Kann nicht mit a beginnen"
END IF
```

IF...THEN...ELSE-Anweisungen können verschachtelt werden; mit anderen Worten kann eine **IF...THEN...ELSE**-Anweisung weitere **IF...THEN...ELSE**-Anweisungen wie folgt enthalten:

```
IF X > 0 THEN
    IF Y > 0 THEN
      IF Z > 0 THEN
         PRINT "Alle größer als Null."
      ELSE
         PRINT "Nur X und Y sind größer als Null."
      END IF
   END IF
ELSEIF X = 0 THEN
   IF Y = 0 THEN
      IF Z = 0 THEN
         PRINT "Alle gleich Null."
      ELSE
         PRINT "Nur X und Y sind gleich Null."
      END IF
   END IF
ELSE
   PRINT "X ist kleiner als Null."
END IF
```

1.3.2 SELECT CASE

Die **SELECT CASE**-Anweisung ist eine der Block-**IF...THEN...ELSE**-Anweisung ähnliche Entscheidungsstruktur mit mehrfachen Auswahlmöglichkeiten. Die Block-**IF...THEN...ELSE** wird genau wie **SELECT CASE** angewandt.

Der entscheidende Unterschied zwischen den beiden besteht darin, daß **SELECT CASE** einen einzigen Ausdruck auswertet und anschließend, je nach Ergebnis, verschiedene Anweisungen ausführt oder zu verschiedenen Teilen des Programmes verzweigt. Eine Block-**IF...THEN...ELSE**, dagegen, kann völlig verschiedene Ausdrücke auswerten.

Beispiele

Die folgenden Beispiele verdeutlichen die Ähnlichkeiten und Unterschiede zwischen **SELECT CASE**- und **IF...THEN...ELSE**-Anweisungen. Das nächste Beispiel stellt die Anwendung einer Block-**IF...THEN...ELSE** bei einer Entscheidung mit mehrfachen Auswahlmöglichkeiten dar:

```
INPUT X
IF X = 1 THEN
   PRINT "eins"
ELSEIF X = 2 THEN
   PRINT "zwei"
ELSEIF X = 3 THEN
   PRINT "drei"
ELSE
   PRINT "muß ganzzahlig zwischen 1-3 sein"
END IF
```

Die oben aufgeführte Entscheidung wurde unter Verwendung von **SELECT CASE** umgeschrieben:

```
INPUT X
SELECT CASE X
   CASE 1
      PRINT "eins"
   CASE 2
      PRINT "zwei"
   CASE 3
      PRINT "drei"
   CASE ELSE
      PRINT "muß ganzzahlig zwischen 1-3 sein"
END SELECT
```

Folgende Entscheidung kann mit Hilfe der **SELECT CASE**, sowie der Block-**IF...THEN...ELSE**-Anweisung bearbeitet werden. Der Vergleich anhand der **IF...THEN...ELSE**-Anweisung ist jedoch zweckmäßiger, da in der **IF**- und **ELSEIF**-Klausel verschiedene Ausdrücke ausgewertet werden:

```
INPUT X, Y
IF X = 0 AND Y = 0 THEN
   PRINT "Beide sind Null."
ELSEIF X = 0 THEN
   PRINT "Nur X ist Null."
ELSEIF Y = 0 THEN
   PRINT "Nur Y ist Null."
ELSE
   PRINT "Keine von beiden ist Null."
END IF
```

1.3.2.1 Anwendung der SELECT CASE-Anweisung

Tabelle 1.3 zeigt die Syntax der **SELECT CASE**-Anweisung und deren Anwendung.

Tabelle 1.3 Syntax von SELECT CASE und Beispiel

Syntax

SELECT CASE *Ausdruck*
 CASE *Ausdrucksliste1*
  [*Anweisungsblock-1*]
 [**CASE** *Ausdrucksliste2*
  [*Anweisungsblock-2*]]
 .
 .
 .
 [**CASE ELSE**
  [*Anweisungsblock-n*]]
END SELECT

Beispiel

```
INPUT TestWert
SELECT CASE TestWert
  CASE 1, 3, 5, 7, 9
   PRINT "ungerade"
  CASE 2, 4, 6, 8
   PRINT "gerade"
  CASE IS < 1
   PRINT "Zu klein"
  CASE IS > 9
   PRINT "Zu groß"
  CASE ELSE
   PRINT "Keine Ganzzahl"
END SELECT
```

Nach einer **CASE**-Klausel können ein oder mehrere der folgenden, durch Kommata getrennte Argumente der *Ausdrucksliste* stehen:

- Ein numerischer Ausdruck oder ein Bereich von numerischen Ausdrücken.
- Ein Zeichenketteausdruck oder ein Bereich von Zeichenketteausdrücken.

Zum Feststellen eines Bereiches von Ausdrücken ist folgende Syntax für die **CASE**-Anweisung zu benutzen:

CASE *Ausdruck* **TO** *Ausdruck*
CASE IS *Vergleichsoperator-Ausdruck*

Der *Vergleichsoperator* ist ein beliebiger Operator der Tabelle 1.1. Benutzen Sie beispielsweise `CASE 1 TO 4`, werden die mit diesem Fall verknüpften Anweisungen ausgeführt, wenn der *Ausdruck* in der **SELECT CASE**-Anweisung größer gleich 1 und kleiner gleich 4 ist. Benutzen Sie `CASE IS < 5`, werden die entsprechenden Anweisungen nur ausgeführt, wenn *Ausdruck* kleiner als 5 ist.

Wird ein Bereich mit dem Schlüsselwort **TO** ausgedrückt, ist der kleinere Wert zuerst einzugeben. Wenn Sie beispielsweise überprüfen möchten, ob ein Wert im Bereich von -5 bis -1 liegt, sollte die **CASE**-Anweisung wie folgt geschrieben werden:

```
CASE -5 TO -1
```

Die folgende Anweisung stellt jedoch keine gültige Möglichkeit zur Festlegung des Bereiches von -5 bis -1 dar, so daß die mit diesem Fall verknüpften Anweisungen nicht ausgeführt werden:

```
CASE -1 TO -5
```

Ähnlich sollte ein Bereich von Zeichenkettekonstanten die Zeichenketten in alphabetischer Reihenfolge listen:

```
CASE "aber" TO "bunt"
```

Für jede **CASE**-Klausel können mehrere Ausdrücke oder Bereiche von Ausdrücken wie in den folgenden Zeilen aufgelistet werden:

```
CASE 1 TO 4, 7 TO 9, JokerZeich1%, JokerZeich2%
CASE IS = Test$, IS = "Ende der Daten"
CASE IS < UnterGr, 5, 6, 12, IS > OberGr
CASE IS < "HAN", "MAO" TO "TAO"
```

Erscheint der Wert des **SELECT CASE**-Ausdruckes in der Liste, die einer **CASE**-Klausel folgt, werden nur die mit dieser **CASE**-Klausel verbundenen Anweisungen ausgeführt. Die Kontrolle springt anschließend zu der ersten ausführbaren Anweisung die **END SELECT** folgt, nicht zu dem nächsten Anweisungsblock innerhalb der **SELECT CASE**-Struktur, wie durch die Ausgabe des nächsten Beispiels veranschaulicht (wenn der Benutzer die Zahl 1 eingibt):

```
INPUT x
SELECT CASE x
   CASE 1
      PRINT "eins, ";
   CASE 2
      PRINT "zwei, ";
   CASE 3
      PRINT "drei, ";
END SELECT
PRINT "das war's"
```

Ausgabe

```
? 1
eins, das war's
```

Erscheint derselbe Wert oder Wertbereich in mehr als einer **CASE**-Klausel, werden nur die mit dem ersten Vorkommen verbundenen Anweisungen ausgeführt, so wie in der nächsten Ausgabe veranschaulicht (wobei der Benutzer "ATLAS" eingegeben hat):

```
INPUT Test$
SELECT CASE Test$
   CASE "A" TO "AZZZZZZZZZZZZZZZZZ"
      PRINT "Ein Wort in Großbuchstaben, das mit A beginnt"
   CASE IS < "A"
      PRINT "Keine alphabetischen Zeichen"
   CASE "ATLAS"
      ' Dieser Fall wird niemals ausgeführt, da das Wort
      ' ATLAS im Bereich der ersten CASE-Klausel liegt;
      PRINT "Ein besonderer Fall"
END SELECT
```

Ausgabe

```
? ATLAS
Ein Wort in Großbuchstaben, das mit A beginnt
```

Wird eine **CASE ELSE**-Klausel benutzt, muß diese als letzte **CASE**-Klausel innerhalb der **SELECT CASE**-Anweisung aufgeführt sein. Die Anweisungen zwischen einer **CASE ELSE**-Klausel und einer **END SELECT**-Klausel werden nur dann ausgeführt, wenn der *Ausdruck* keiner anderen **CASE**-Auswahl in der **SELECT CASE**-Anweisung entspricht. Es ist zum Beispiel empfehlenswert, eine **CASE ELSE**-Anweisung in den **SELECT CASE**-Block einzufügen, um auf unvorhergesehene Werte für *Ausdruck* reagieren zu können. Ist jedoch keine **CASE ELSE**-Anweisung vorhanden und entspricht *Ausdruck* keiner **CASE**-Auswahl, setzt das Programm die Ausführung fort.

Beispiel

Der folgende Programmausschnitt zeigt, wie die **SELECT CASE**-Anweisung am häufigsten verwendet wird.Es wird ein Menü auf dem Bildschirm ausgegeben und verzweigt dann ausgehend von der vom Benutzer eingegebenen Zahl in verschiedene Unterprogramme.

```
DO                          ' Starte Menüschleife
   CLS                      ' Bildschirm löschen
   ' Schreibe fünf Menüpunkte auf den Bildschirm:
   PRINT "HAUPTMENÜ" : PRINT
   PRINT "1)  Hinzufügen neuer Namen"
   PRINT "2)  Löschen von Namen"
   PRINT "3)  Informationen verändern"
   PRINT "4)  Auflisten der Namen"
   PRINT
   PRINT "5)  ENDE"
   ' Gib Eingabeaufforderung aus:
   PRINT : PRINT "Geben Sie Ihre Wahl ein (1 bis 5):"
   ' Warte auf die Eingabe des Benutzers. INPUT$(1)
   ' Lies ein eingegebenes Zeichen von der Tastatur:
   Bu$ = INPUT$(1)
   ' Benutze SELECT CASE zu der Verarbeitung der Antwort:
   SELECT CASE Bu$
        CASE "1"
           CALL HinzDaten
        CASE "2"
           CALL LoeschDat
        CASE "3"
           CALL VerDaten
        CASE "4"
           CALL ListDaten
        CASE "5"
           EXIT DO           ' Die einzige Möglichkeit die
                             ' Menüschleife zu verlassen.
        CASE ELSE
           BEEP
   END SELECT
LOOP                        ' Ende der Menüschleife
END
' Unterprogramme HinzDaten, LoeschDat, VerDaten und
' ListDaten:
.
.
.
```

1.3.2.2 SELECT CASE im Vergleich mit ON...GOSUB

Die vielseitigere **SELECT CASE**-Anweisung kann als Ersatz für den alten **ON** *Ausdruck* {**GOSUB** | **GOTO**} verwendet werden. Nachstehend werden die Vorteile der **SELECT CASE**-Anweisung gegenüber der **ON...GOSUB**-Anweisung zusammengefaßt:

- Der *Ausdruck* in **SELECT CASE** *Ausdruck* kann sowohl einen Zeichenkettenwert als auch einen numerischen Wert liefern. Der in der Anweisung **ON** *Ausdruck* {**GOSUB** | **GOTO**} angegebene Ausdruck darf jedoch nur einen numerischen Wert im Bereich von 0 bis 255 haben.
- Die **SELECT CASE**-Anweisung verzweigt direkt zu einem Anweisungsblock, der unmittelbar nach der passenden **CASE**-Klausel steht. **ON** *Ausdruck* **GOSUB** verzweigt dagegen zu einer Unterroutine in einem anderen Programmteil.
- **CASE**-Klauseln können dazu verwendet werden, einen *Ausdruck* zu überprüfen, ob er in einem Bereich von Werten liegt. Wie aus dem nachfolgenden Beispiel hervorgeht, kann dies mit **ON** *Ausdruck* {**GOSUB** | **GOTO**} nur schwer erreicht werden, besonders wenn es sich um einen großen Bereich handelt.

Im folgenden Ausschnitt verzweigt die **ON...GOSUB**-Anweisung je nach dem Wert der Benutzereingabe in eine der Unterroutinen 50, 100 oder 150.

```
X% = -1
WHILE X%
   INPUT "Wählen Sie (0 für Ende): ", X%
   IF X% = 0 THEN END
   WHILE X% < 1 OR X% > 12
      PRINT "Es muß ein Wert von 1 bis 12 sein"
      INPUT "Wählen Sie (0 für Ende): ", X%
   WEND
   ON x% GOSUB 50,50,50,50,50,50,50,100,100,100,150
WEND
.
.
.
```

Vergleichen Sie das obige Beispiel mit dem nächsten, das eine **SELECT CASE**-Anweisung mit Wertebereichen in jeder **CASE**-Klausel verwendet:

```
DO
   INPUT "Wählen Sie (0 für Ende): ", X%
   SELECT CASE X%
      CASE 0
         END
      CASE 1 TO 8        ' Ersetzt "Unterroutine 50" im
                         ' vorhergehenden Beispiel
```

```
        CASE 9 TO 11      ' Ersetzt "Unterroutine 100" im
                          ' vorhergehenden Beispiel
        CASE 12           ' Ersetzt "Unterroutine 150" im
                          ' vorhergehenden Beispiel
        CASE ELSE         ' Eingabe war außerhalb des
                          ' Bereiches.
           PRINT "Es muß ein Wert von 1 bis 12 sein"
        END SELECT
LOOP
```

1.4 Schleifenstrukturen

Schleifenstrukturen wiederholen einen Block von Anweisungen (die Schleife) entweder sooft wie angegeben oder bis ein bestimmter Ausdruck (die Schleifenbedingung) wahr oder falsch ist.

Benutzer von BASICA sind mit den zwei Schleifenstrukturen, **FOR...NEXT** und **WHILE...WEND**, vertraut, die in den Abschnitten 1.4.1 und 1.4.2 erläutert werden. Mit der **DO...LOOP**-Anweisung bietet QuickBASIC eine zusätzliche Schleifenstruktur. Sie wird in Abschnitt 1.4.3 erläutert.

1.4.1 FOR...NEXT-Schleifen

Eine **FOR...NEXT**-Schleife wiederholt die in der Schleife eingeschlossenen Anweisungen sooft wie angegeben. Sie zählt von einem Startwert bis zu einem Endwert, indem ein Schleifenzähler entweder erhöht oder vermindert wird. Solange der Schleifenzähler den Endwert nicht erreicht hat, fährt die Schleife mit der Ausführung fort. Tabelle 1.4 stellt die Syntax der **FOR...NEXT**-Anweisung und ein Beispiel ihrer Verwendung dar:

Tabelle 1.4 FOR...NEXT-Syntax und Beispiel

Syntax	***Beispiel***

FOR *Zähler* = *Start* **TO** *Ende* [**STEP** *Schrittgröße*]
[*Anweisungsblock-1*]
[**EXIT FOR**
[*Anweisungsblock-2*]]
NEXT [*Zähler*]
	`FOR I% = 1 bis 10`
	` DatFeld%(I%) = I%`
	`NEXT`

In einer **FOR...NEXT**-Schleife hat die Zähler-Variable ursprünglich den Wert des Ausdrucks *Start*. Nach jedem Durchlauf der Schleife wird der Wert von *Zähler* angepaßt. Wird das Schlüsselwort **STEP** ausgelassen, beträgt der Standardwert für die Anpassung eins; das bedeutet, daß dem *Zähler* bei jeder Ausführung der Schleife der Wert Eins hinzugezählt wird. Bei Verwendung des Schlüsselwortes **STEP** wird der Zähler durch den Betrag von *Schrittgröße* angepaßt. Das Argument *Schrittgröße* kann jeden numerischen Wert annehmen; wenn es negativ ist, zählt die Schleife abwärts von *Start* bis *Ende*. Nach Erhöhen oder Vermindern der *Zähler*-Variable wird deren Wert mit *Ende* verglichen. Die Schleife ist an diesem Punkt beendet, wenn einer der folgenden Punkte wahr ist:

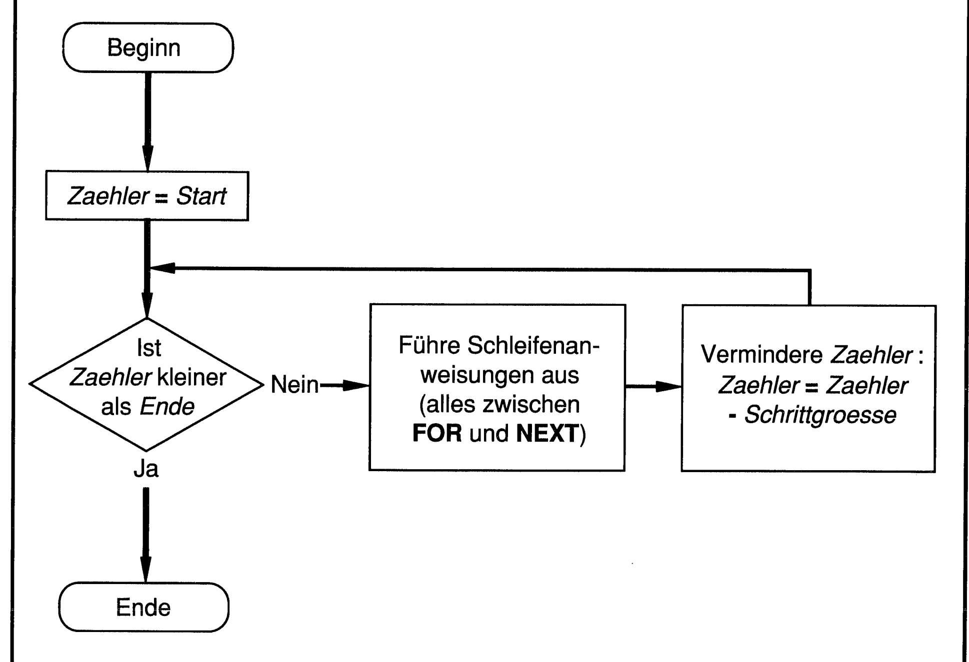
- Die Schleife wird hochgezählt (*Schrittgröße* ist positiv) und *Zähler* ist größer als *Ende*.
- Die Schleife wird abwärts gezählt (*Schrittgröße* ist negativ) und *Zähler* ist kleiner als *Ende*.

Abbildung 1.1 zeigt die Logik einer **FOR...NEXT**-Schleife, wenn der Wert von *Schrittgröße* positiv ist.

Abbildung 1.1 Logik einer FOR...NEXT-Schleife mit positiver STEP-Größe

Abbildung 1.2 zeigt die Logik einer **FOR...NEXT**-Schleife, wenn der Wert von *Schrittgröße* negativ ist.

Abbildung 1.2 Logik einer FOR...NEXT Schleife mit negativer STEP-Größe



Eine **FOR...NEXT**-Anweisung "überprüft immer am oberen Ende der Schleife", so daß die Schleife nie ausgeführt wird, wenn eine der folgenden Bedingungen wahr ist:

- Die Schrittgröße ist positiv und der Anfangswert von *Start* ist größer als der Wert von *Ende*:

```
' Schleife wird nie ausgeführt, da der Anfangswert von I%
' größer als 9 ist:
FOR I% = 10 TO 9
   .
   .
   .
NEXT I%
```

- Die Schrittgröße ist negativ und der Anfangswert von *Start* ist kleiner als der Wert von *Ende*:

```
' Schleife wird nie ausgeführt, da der Anfangswert von I%
' kleiner als 9 ist:
FOR I% = -10 TO -9 STEP -1
   .
   .
   .
NEXT I%
```

Es ist nicht erforderlich, das Argument *Zähler* in der **NEXT**-Klausel zu verwenden. Sind jedoch mehrere verschachtelte **FOR...NEXT**-Schleifen (eine Schleife innerhalb einer anderen) vorhanden, kann das Aufführen der Argumente *Zähler* hilfreich sein, um zu Erkennen in welcher Schleife Sie sich befinden.

Nachstehend werden einige allgemeine Regeln zur Verschachtelung von **FOR...NEXT**-Schleifen aufgeführt:

- Wenn eine Schleifenzählervariable in einer **NEXT**-Klausel benutzt wird, muß der Zähler einer verschachtelten Schleife vor dem Zähler der umschließenden Schleife stehen. Mit anderen Worten ist folgende Verschachtelung zulässig:

```
FOR I = 1 TO 10
   FOR J = -5 TO 0
   .
   .
   .
   NEXT J
NEXT I
```

  Nachstehende Verschachtelung ist jedoch nicht zulässig:

```
FOR I = 1 TO 10
   FOR J = -5 TO 0
   .
   .
   .
   NEXT I
NEXT J
```

- Wenn eine separate **NEXT**-Klausel zum Beenden jeder Schleife verwendet wird, muß die Anzahl der **NEXT**-Klauseln mit der Anzahl der **FOR**-Klauseln übereinstimmen.
- Wenn eine einzelne **NEXT**-Klausel zum Beenden verschiedener Ebenen von **FOR...NEXT**-Schleifen verwendet wird, müssen die Schleifenzählervariablen nach der **NEXT**-Klausel in der Reihenfolge "von innen nach außen" erscheinen:

  NEXT *innerster Schleifenzähler, ..., äußerster Schleifenzähler*

  In diesem Fall muß die Anzahl der Schleifenzählervariablen in der **NEXT**-Klausel der Anzahl der **FOR**-Klauseln entsprechen.

Beispiele

Die drei folgenden Programmausschnitte veranschaulichen verschiedene Möglichkeiten des Verschachtelns von **FOR...NEXT**-Schleifen, um die gleiche nachstehende Ausgabe zu erzeugen. Das erste Beispiel zeigt verschachtelte **FOR...NEXT**-Schleifen mit Schleifenzählern und separaten **NEXT**-Klauseln für jede Schleife:

```
FOR I = 1 TO 2
   FOR J = 4 TO 5
      FOR K = 7 TO 8
         PRINT I, J, K
      NEXT K
   NEXT J
NEXT I
```

Das nächste Beispiel benutzt ebenfalls Schleifenzähler, jedoch nur eine **NEXT**-Klausel für alle drei Schleifen:

```
FOR I = 1 TO 2
   FOR J = 4 TO 5
      FOR K = 7 TO 8
         PRINT I, J, K
NEXT K, J, I
```

Das letzte Beispiel zeigt verschachtelte **FOR...NEXT**-Schleifen ohne Schleifenzähler:

```
FOR I = 1 TO 2
   FOR J = 4 TO 5
      FOR K = 7 TO 8
         PRINT I, J, K
      NEXT
   NEXT
NEXT
```

Ausgabe

```
1      4      7
1      4      8
1      5      7
1      5      8
2      4      7
2      4      8
2      5      7
2      5      8
```

1.4.1.1 Das Verlassen einer FOR...NEXT-Schleife anhand von EXIT FOR

Es kann vorkommen, daß Sie eine **FOR...NEXT**-Schleife verlassen möchten, bevor die Zählervariable den Endwert der Schleife erreicht hat. Dies wird von der **EXIT FOR**-Anweisung ermöglicht. Eine einzige **FOR...NEXT**-Schleife kann eine beliebige Anzahl von **EXIT FOR**-Anweisungen enthalten, wobei die **EXIT FOR**-Anweisungen an beliebiger Stelle innerhalb der Schleife stehen können. Der folgende Ausschnitt stellt eine Verwendungsmöglichkeit der **EXIT FOR**-Anweisung dar:

```
' Schreibe die Quadratwurzeln der Zahlen von 1 bis 30.000.
' Wenn der Benutzer eine Taste während der
' Schleifenausführung betätigt,
' verläßt die Kontrolle die Schleife:
FOR I% = 1 TO 30000
   PRINT SQR (I%)
   IF INKEY$ <> "" THEN EXIT FOR
NEXT
.
.
.
```

EXIT FOR verläßt nur die innerste eingeschlossene **FOR NEXT**-Schleife, in der sie vorkommt. Betätigt beispielsweise der Benutzer während der Ausführung der darauf folgenden verschachtelten Schleifen eine Taste, verläßt das Programm nur die innerste Schleife. Wenn die äußere Schleife immer noch aktiv wäre (das heißt, der Wert von `I% <= 100`), würde die innerste Schleife die Steuerung übernehmen:

```
FOR I% = 1 TO 100
   FOR J% = 1 TO 100
      PRINT I% / J%
      IF INKEY$ <> "" THEN EXIT FOR
   NEXT J%
NEXT I%
```

1.4.1.2 Verzögerung der Programmausführung anhand einer FOR...NEXT-Schleife

Viele BASICA-Programme verwenden eine leere **FOR...NEXT**-Schleife wie die folgende, um in einem Programm eine Verzögerung einzufügen:

```
.
.
.
' Es befinden sich keine Anweisungen in dieser Schleife
' alles was sie tut, ist von 1 bis 10.000 unter Verwendung
' von Ganzzahlen zu zählen.
FOR I% = 1 TO 10000: NEXT
.
.
.
```

Für sehr kurze Verzögerungen oder Verzögerungen, die kein exaktes Intervall bilden müssen, eignet sich die Verwendung von **FOR...NEXT** hervorragend. Das Problem beim Verwenden leerer **FOR...NEXT**-Schleifen in dieser Art und Weise ist, daß verschiedene Computer, verschiedene BASIC-Versionen und verschiedene Compiler-Optionen beeinflussen, wie schnell die Berechnungen in einer **FOR...NEXT**-Schleife ausgeführt werden. Daraus folgt, daß die Dauer einer Verzögerung sehr unterschiedlich sein kann. Dafür bietet jetzt die **SLEEP**-Anweisung von QuickBASIC eine bessere Alternative. (Die Syntax der **SLEEP**-Anweisung ist Kapitel 8, "Zusammenfassung der Anweisungen und Funktionen" zu entnehmen.)

1.4.2 WHILE...WEND-Schleifen

Die **FOR...NEXT**-Anweisung ist zweckmäßig, wenn die Anzahl der Schleifenausführungen bereits vorher bekannt ist. Wenn jedoch die Anzahl der Schleifenausführungen unvorhersehbar, aber die Bedingung bekannt ist, die die Schleife abbricht, kann die **WHILE...WEND**-Anweisung vorteilhaft angewandt werden. Anstatt per Zählung festzustellen, ob eine Schleife weiterhin ausgeführt werden soll, wiederholt **WHILE...WEND** die Schleife solange die Schleifenbedingung wahr ist.

Tabelle 1.5 zeigt die Syntax der **WHILE...WEND**-Anweisung und ein Beispiel ihrer Anwendung:

Tabelle 1.5 Syntax der WHILE...WEND-Anweisung und Beispiel

Syntax	***Beispiel***

WHILE *Bedingung*
[*Anweisungsblock*]
WEND
	`INPUT R$`
	`WHILE R$ <> "J" AND R$ <> "N"`
	`  PRINT "Antwort muß J oder N sein."`
	`  INPUT R$`
	`WEND`

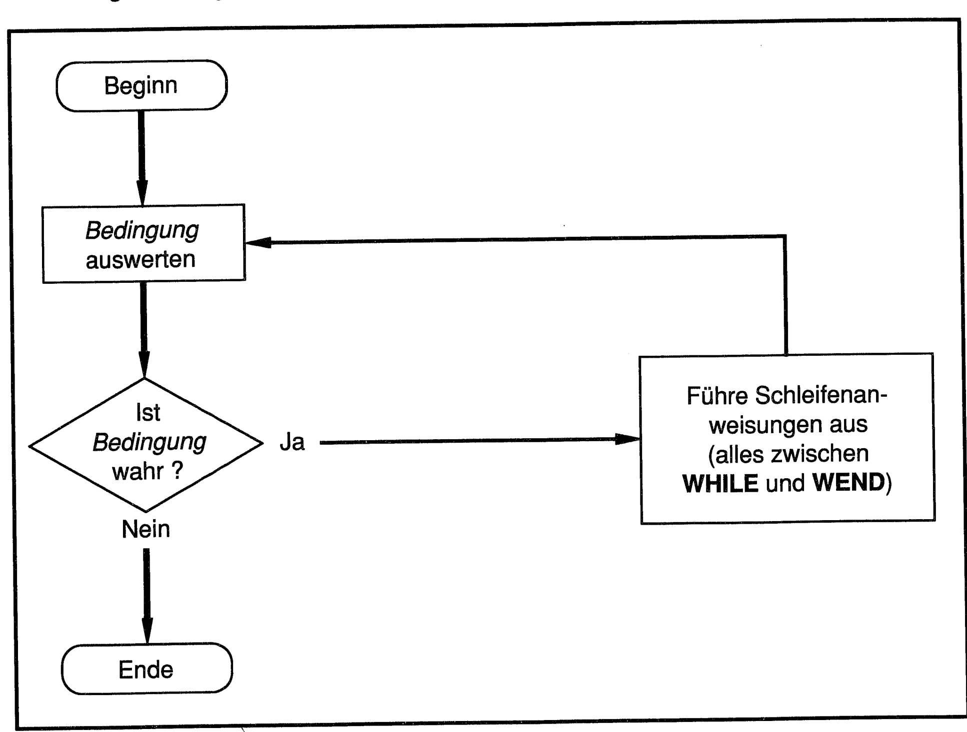
Beispiel

Das folgende Beispiel weist der Variablen X einen Anfangswert von zehn zu und halbiert diesen solange, bis der Wert von X kleiner als 0,00001 ist:

```
X = 10
WHILE X > 0,00001
   PRINT X
   X = X/2
WEND
```

Abbildung 1.3 veranschaulicht die Logik einer **WHILE...WEND**-Schleife.

Abbildung 1.3 Logik einer WHILE...WEND-Schleife



1.4.3 DO...LOOP-Schleifen

Wie die **WHILE...WEND**-Anweisung, führt auch die **DO...LOOP**-Anweisung einen Anweisungsblock unbestimmt oft aus; das bedeutet, daß das Verlassen der Schleife davon abhängt, ob die Schleifenbedingung falsch oder wahr ist. Zum Unterschied von **WHILE...WEND** erlaubt Ihnen **DO...LOOP** auf eine wahre oder eine falsche Bedingung hin zu prüfen. Zudem kann die Prüfung der Bedingung sowohl am Anfang als auch am Ende der Schleife stehen.

Tabelle 1.6 zeigt die Syntax einer Schleife, die die Prüfung am Anfang vornimmt.

Tabelle 1.6 DO...LOOP-Syntax und Beispiel: Prüfung am Anfang

Syntax	*Beispiel*

DO [{**WHILE** | **UNTIL**}*Bedingung*]
[*Anweisungsblock-1*]
[EXIT DO
[*Anweisungsblock-2*]]
LOOP
	`DO UNTIL INKEY$ <> ""`
	`' Eine leere Schleife die`
	`' verzögert bis eine Taste`
	`' betätigt wird`
	`LOOP`

Abbildungen 1.4 und 1.5 veranschaulichen die beiden Arten von **DO...LOOP**, die die Prüfung am Anfang der Schleife vornehmen.

Abbildung 1.4 Logik einer DO WHILE...LOOP

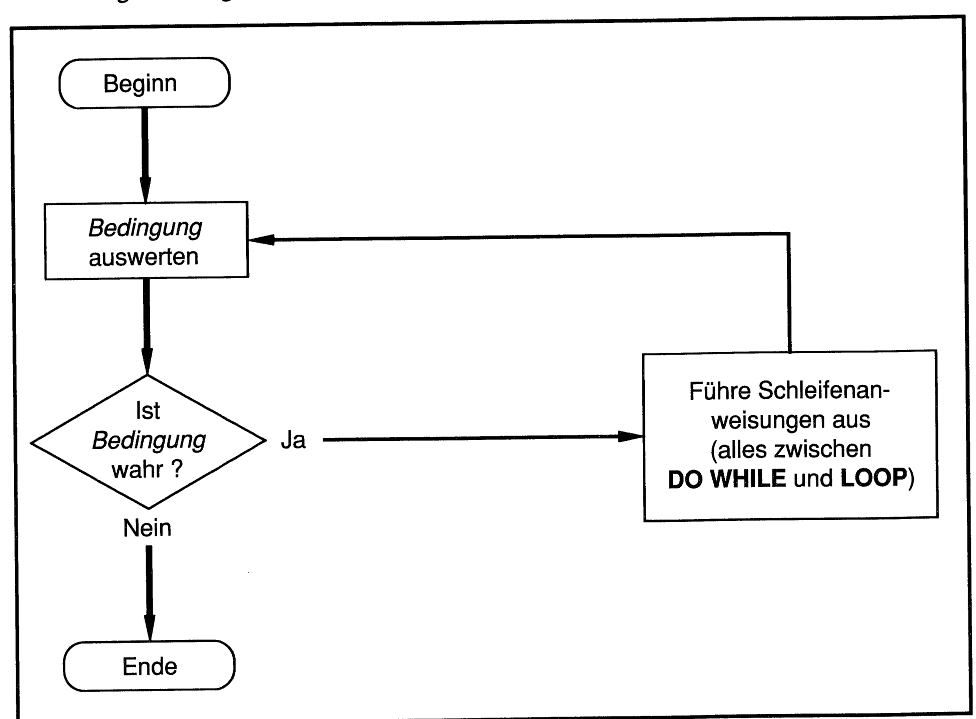


Abbildung 1.5 Logik einer DO UNTIL...LOOP

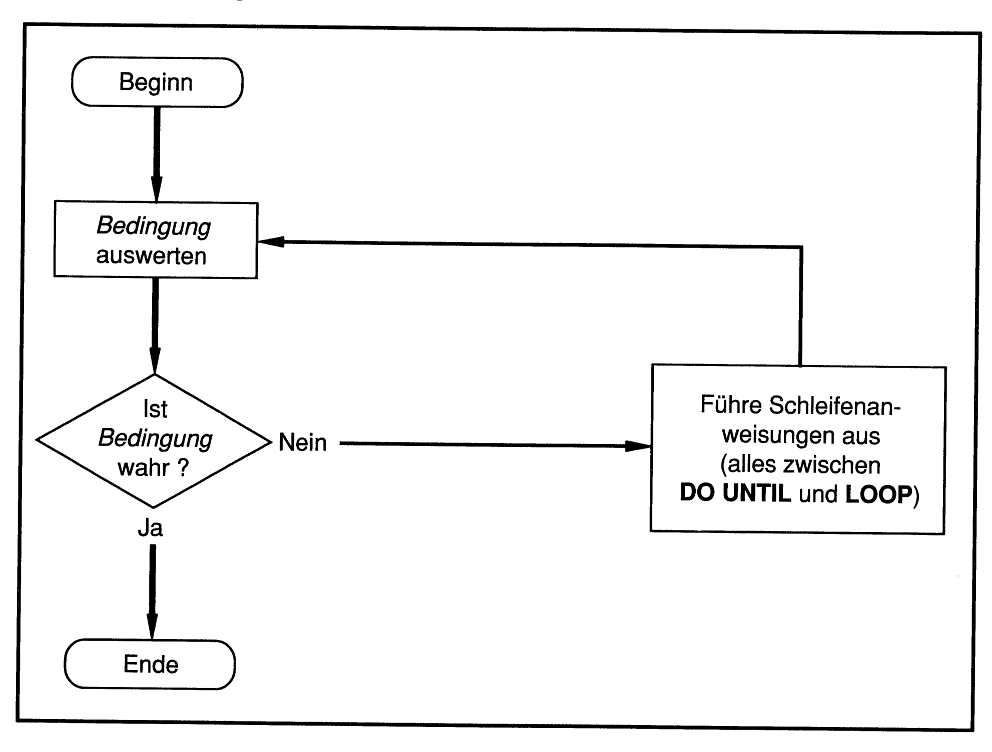


Tabelle 1.7 zeigt die Syntax einer Schleife, die am Ende der Schleife auf wahr oder falsch prüft.

Tabelle 1.7 DO...LOOP-Syntax und Beispiel: Prüfung am Ende

Syntax

DO
 [*Anweisungsblock-1*]
 [EXIT DO
 [*Anweisungsblock-2*]]
LOOP WHILE [{**WHILE** | **UNTIL**}*Bedingung*]

Beispiel

```
DO
 INPUT "Verändern:", Ve
 Gesamt = Gesamt + Ve
LOOP WHILE Ve <> 0
```

Abbildung 1.6 und 1.7 stellen die beiden Arten der **DO...LOOP** dar, die am Ende der Schleife prüfen.

Abbildung 1.6 Logik von DO...LOOP WHILE

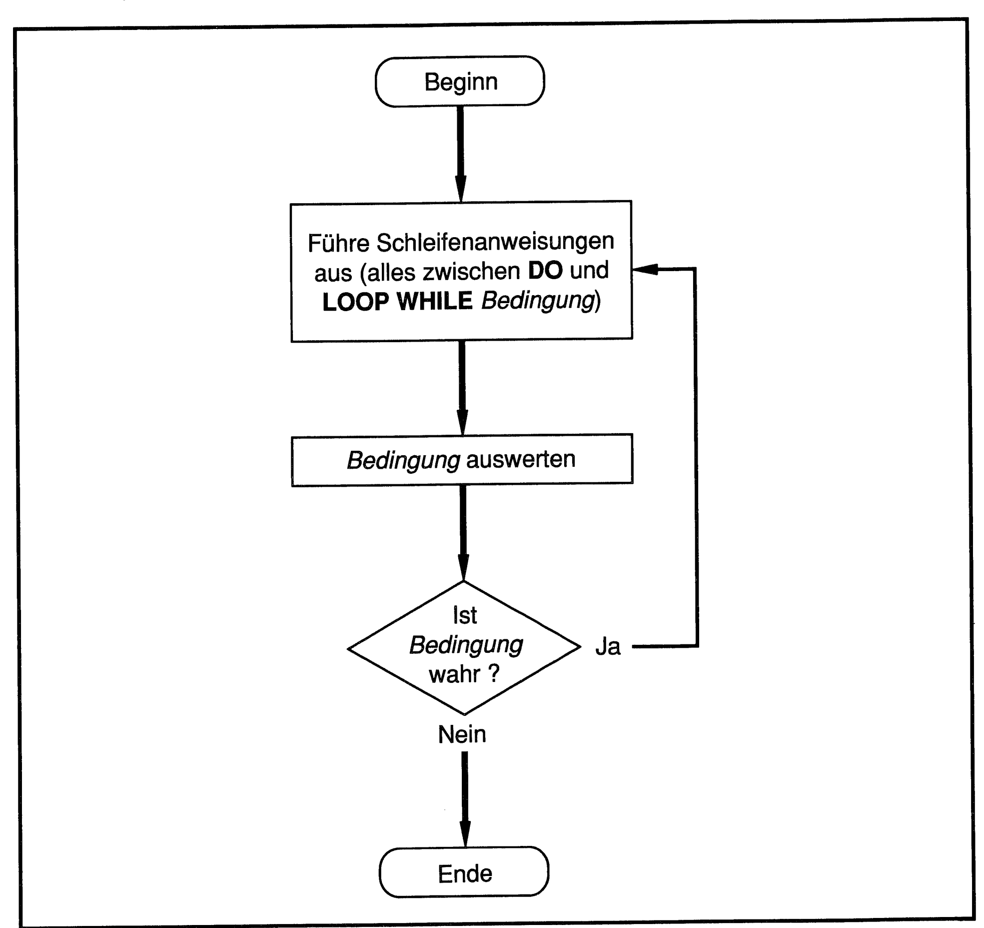
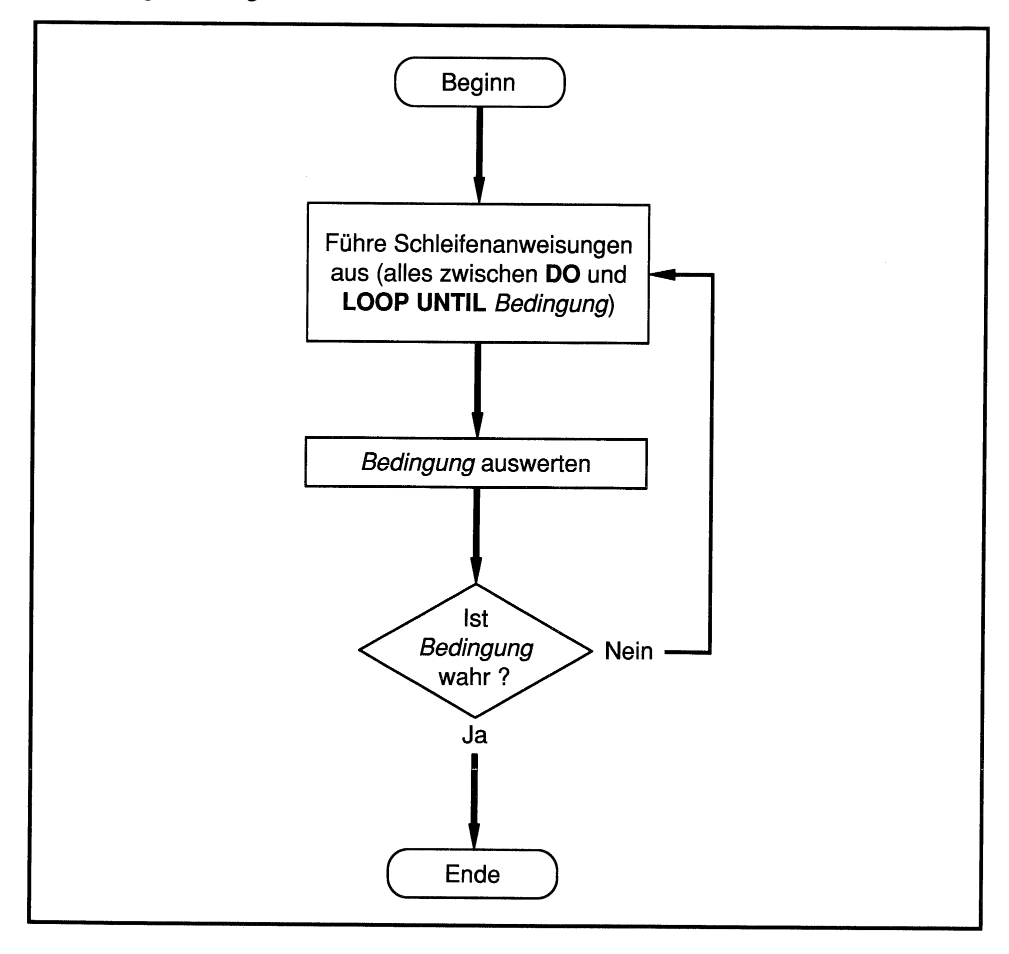


Abbildung 1.7 Logik von DO...LOOP UNTIL



1.4.3.1 Schleifenprüfungen: eine Möglichkeit, DO...LOOP zu verlassen

Am Ende einer **DO...LOOP**-Anweisung läßt sich anhand einer Schleifenprüfung eine Schleife erstellen, in der die Anweisungen mindestens einmal ausgeführt werden. Bei der **WHILE...WEND**-Anweisung muß manchmal auf den Programmiertrick der Voreinstellung der Schleifenvariable auf einen beliebigen Wert zurückgegriffen werden, um den ersten Schleifendurchlauf zu erzwingen. Mit **DO...LOOP** sind solche Tricks überflüssig, so wie aus den nachfolgenden Beispielen ersichtlich, die beide Verfahren veranschaulichen:

```
' WHILE...WEND-Schleife prüft am Anfang, so daß die
' Zuweisung von "J" an
' Antwort$ erforderlich ist, um mindestens eine Ausführung
' der Schleife zu erzwingen
```

```
Antwort$ = "J"
WHILE UCASE$ (Antwort$) = "J"
   .
   .
   .
   INPUT "Noch einmal"; Antwort$
WEND
' Dieselbe Schleife mit DO...LOOP zur Überprüfung am Ende
' der Schleife:
DO
   .
   .
   .
   INPUT "Noch einmal"; Antwort$
LOOP WHILE UCASE$ (Antwort$) = "Y"
```

Eine durch **WHILE** ausgedrückte Bedingung kann anhand von **UNTIL** wie folgt umgeschrieben werden:

```
' ============================================================
'             Verwendung von DO WHILE NOT...LOOP
' ============================================================
' Solange das Ende der Datei 1 nicht erreicht ist, lies eine
' Zeile aus der Datei und schreibe sie auf den Bildschirm:
DO WHILE NOT EOF (1)
   LINE INPUT #1, ZeilenPuffer$
   PRINT ZeilenPuffer$
LOOP
' ============================================================
'                 Verwendung von DO UNTIL LOOP
' ============================================================
' Bis das Ende der Datei 1 erreicht ist, lies eine Zeile aus
' der Datei und schreibe sie auf den Bildschirm:
DO UNTIL EOF (1)
   LINE INPUT #1, ZeilenPuffer$
   PRINT ZeilenPuffer$
LOOP
```

1.4.3.2 EXIT DO: Eine Alternative, DO...LOOP zu verlassen

Innerhalb einer **DO...LOOP**-Anweisung werden andere Anweisungen ausgeführt, die letztlich die Schleifenprüfungsbedingung von wahr auf falsch oder von falsch auf wahr verändern, so daß die Schleife beendet wird. In den bisher aufgeführten Beispielen zu **DO...LOOP** wurde die Prüfung entweder am Anfang oder am Ende der Schleife vorgenommen. Wird jedoch die Schleife durch die **EXIT DO**-Anweisung verlassen, läßt sich die Prüfung innerhalb der Schleife plazieren. Eine einzelne **DO...LOOP**-Schleife kann eine beliebige Anzahl von **EXIT DO**-Anweisungen enthalten, und die **EXIT DO**-Anweisungen können an beliebiger Stelle innerhalb der Schleife erscheinen.

Beispiel

Im folgenden Beispiel wird eine Datei geöffnet und Zeile für Zeile bis zum Dateiende oder bis zum Erreichen des vom Benutzer eingegebenen Musters eingelesen. Wird das Muster vor Erreichen des Dateiendes entdeckt, so wird die Schleife anhand einer **EXIT DO**-Anweisung verlassen.

```
INPUT "Zu durchsuchende Datei: ", Datei$
IF Datei$ = "" THEN END
INPUT "Zu suchendes Muster: ", Muster$
OPEN Datei$ FOR INPUT AS #1
DO UNTIL EOF(1)      ' EOF(1) gibt einen wahren Wert an, wenn
das Dateiende
                     ' erreicht ist.
   LINE INPUT #1, TempZeile$
   IF INSTR(TempZeile$, Muster$) > 0 THEN
      ' Schreibe die erste Zeile, die das Muster enthält und
      ' verlasse die Schleife:
      PRINT TempZeile$
      EXIT DO
   END IF
LOOP
```

1.5 Anwendungsbeispiele

Die Anwendungsbeispiele in diesem Kapitel bestehen aus einem Programm zur Scheckheftsaldierung und einem Programm, das sicherstellt, daß jede Zeile in einer Textdatei mit einer Sequenz Wagenrücklauf-Zeilenvorschub endet.

1.5.1 Programm zur Scheckheftsaldierung (*scheck.bas*)

Dieses Programm fordert den Benutzer auf, den ursprünglichen Kontostand des Girokontos und alle vorgenommenen Transaktionen - Abhebungen oder Einzahlungen - einzugeben. Das Programm gibt dann eine sortierte Liste der Transaktionen und den Schlußkontostand aus.

Verwendete Anweisungen und Funktionen

Dieses Programm wendet folgende in diesem Kapitel erläuterte Anweisungen an:

- **DO...LOOP WHILE**
- **FOR...NEXT**
- **EXIT FOR**
- Block-**IF...THEN...ELSE**

Programm-Listing

```
DIM Betrag(1 TO 100)
CONST FALSCH = 0, WAHR = NOT FALSCH
' Hole die Anfangsbilanz des Kontos:
CLS
PRINT "Geben Sie die Anfangsbilanz ein, und betätigen Sie";
PRINT " die Eingabetaste";
INPUT ":", Bilanz
' Hole die Transaktionen. Fahre mit der Annahme der
' Transaktionen fort, bis null für eine Transaktion
' eingegeben wird, oder bis 100 Transaktionen eingegeben
' sind:
FOR TransNum% = 1 TO 100
   PRINT TransNum%;
   PRINT ") Eingabe des Bewegungsbetrages "
   PRINT "(0 zum beenden): ";
   INPUT "", Betrag(TransNum%)
   IF Betrag(TransNum%) = 0 THEN
      TransNum% = TransNum% - 1
      EXIT FOR
   END IF
NEXT
' Sortiere Transaktionen in aufsteigender Reihenfolge
' unter Verwendung eines "Bubblesort":
Limit% = TransNum%
```

```
DO
   Tausch% = FALSCH
   FOR I% = 1 TO (Limit%-1)
      ' Wenn zwei aufeinanderfolgende Elemente nicht
      ' in der richtigen Reihenfolge sind, tausche
      ' diese Elemente:
      IF Betrag(I%) < Betrag(I%+1) THEN
         SWAP Betrag(I%), Betrag(I%+1)
         Tausch% = I%
      END IF
   NEXT I%
   ' Sortiere im nächsten Schritt nur bis dahin, wo
   ' der letzte Tausch vorgenommen wurde:
   IF Tausch% THEN Limit% = Tausch%
' Sortiere, bis keine Elemente mehr getauscht sind:
LOOP WHILE Tausch%
' Gib das sortierte Datenfeld der Transaktionen aus. Wenn
' eine Bewegung größer als null ist, gib sie als
' "EINZAHLUNG" aus, wenn sie kleiner als null ist, gib
' sie als "AUSZAHLUNG" aus:
FOR I% = 1 TO TransNum%
   IF Betrag(I%) > 0 THEN
      PRINT USING "EINZAHLUNG: DM#####,##"; Betrag(I%)
   ELSEIF Betrag(I%) < 0 THEN
      PRINT USING "AUSZAHLUNG: DM#####,##"; Betrag(I%)
   END IF
   ' Aktualisiere Bilanz:
   Bilanz = Bilanz + Betrag(I%)
NEXT I%
' Gib die Schlußbilanz aus:
PRINT
PRINT "--------------------------"
PRINT USING "Endergebnis: DM######,##"; Bilanz
END
```

1.5.2 Filter für Wagenrücklauf-Zeilenvorschub (*wrzv.bas*)

Einige Textdateien sind in einem Format gespeichert, das nur einen Wagenrücklauf (Rücklauf zum Anfang der Zeile) oder einen Zeilenvorschub (Vorschub auf die nächste Zeile) zum Anzeigen eines Zeilenendes verwendet. Zahlreiche Texteditoren erweitern diesen einzelnen Wagenrücklauf (WR) oder Zeilenvorschub (ZV) in eine Sequenz Wagenrücklauf-Zeilenvorschub (WR-ZV) beim Laden der Datei zum Editieren. Wird jedoch ein Texteditor verwendet, der einen einzelnen WR oder ZV nicht in einem WR-ZV erweitert, muß die Datei möglicherweise so verändert werden, daß sie am Ende jeder Zeile die richtige Sequenz aufweist.

Das folgende Programm bildet einen Filter, der eine Datei öffnet, einen einzelnen WR oder ZV in eine Kombination WR-ZV erweitert und anschließend die ausgerichtete Zeile in eine neue Datei schreibt. Der ursprüngliche Inhalt der Datei wird in einer Datei mit der Namenserweiterung *.bak* gespeichert.

Verwendete Anweisungen und Funktionen

Dieses Programm wendet folgende in diesem Kapitel erläuterte Anweisungen an:

- **DO...LOOP WHILE**
- **DO UNTIL...LOOP**
- Block-**IF...THEN...ELSE**
- **SELECT CASE...END SELECT**

Um dieses Programm zweckmäßiger zu gestalten, enthält es ebenfalls folgende, in diesem Kapitel nicht erläuterte Konstruktionen:

- Eine **FUNCTION**-Prozedur mit dem Namen `Sicherung$`, die eine Datei mit der Erweiterung *.bak* anlegt.

  Weitere Informationen zur Definition und Verwendung von Prozeduren entnehmen Sie bitte Kapitel 2, "Prozeduren: Unterprogramme und Funktionen".
- Eine Fehlerbehandlungsroutine mit dem Namen `FehlerBehand`, um Fehler zu behandeln, die bei der Eingabe eines Dateinamens durch den Benutzer auftreten können. Wenn der Benutzer beispielsweise den Namen einer nicht existierenden Datei eingegeben hat, fragt die Routine erneut nach einem Namen. Ohne diese Routine würde ein solcher Fehler das Programm beenden.

  Weitere Informationen zur Verfolgung von Fehlern finden Sie in Kapitel 6, "Fehler- und Ereignisverfolgung".

Programm-Listing

```
DEFINT A-Z          'Standardvariablentyp ist Ganzzahl.
' Die FUNCTION Sicherung$ erstellt eine
' Sicherungsdatei mit dem Basisnamen der
' zu verändernden Datei plus einer
' der Namenserweiterung .bak:
DECLARE FUNCTION Sicherung$ (DateiName$)
' Initialisiere symbolische Konstanten und Variablen:
CONST FALSCH = 0, WAHR = NOT FALSCH
DruRueck$ = CHR$(13)
ZeilenVor$ = CHR$(10)
DO
   CLS
   ' Eingabe des Namens der zu verändernden Datei:
   INPUT "Welche Datei wollen Sie übertragen"; AusDatei$
   InDatei$ = Sicherung$(AusDatei$) ' Hole den Namen
                                    ' der
                                    ' Sicherungsdatei.
   ON ERROR GOTO FehlerBehand  ' Schalte
                               ' Fehlerverfolgung ein.
   NAME AusDatei$ AS InDatei$  ' Kopiere die
                               ' Eingabedatei in die
                               ' Sicherungsdatei.
   ON ERROR GOTO 0             ' Schalte die
                               ' Fehlerverfolgung aus.
   ' Öffne die Sicherungsdatei für Eingabe und die
   ' alte Datei für Ausgabe:
   OPEN InDatei$ FOR INPUT AS #1
   OPEN AusDatei$ FOR OUTPUT AS #2
   ' Die Variable VorDruRueck ist eine Flag, die
   ' immer dann auf wahr gesetzt wird, wenn das
   ' Programm ein Wagenrücklaufzeichen liest:
   VorDruRueck = FALSCH
   ' Lies aus der Eingabedatei bis das Dateiende
   ' erreicht ist:
   DO UNTIL EOF(1)
      ' Dies ist nicht das Ende der Datei, lies
      ' deshalb ein Zeichen:
      DateiZeich$ = INPUT$(1, #1)
```

```
    SELECT CASE DateiZeich$
        CASE DruRueck$      ' Das Zeichen ist ein WR.
           ' Wenn das vorhergehende Zeichen
           ' ebenfalls ein WR war, schreibe ein ZV
           ' vor das Zeichen:
           IF VorDruRueck THEN
              DateiZeich$ = ZeilenVor$ + DateiZeich$
           END IF
           ' Setze in jedem Fall die Variable
           ' VorDruRueck auf WAHR:
           VorDruRueck = WAHR
        CASE ZeilenVor$     ' Das Zeichen ist ein ZV.
           ' Wenn das vorhergehende Zeichen kein WR
           ' war, schreibe ein WR vor das Zeichen:
           IF NOT VorDruRueck THEN
              DateiZeich$ = DruRueck$ + DateiZeich$
           END IF
           ' Setze die Variable VorDruRueck in jedem
           ' Fall auf FALSCH:
           VorDruRueck = FALSCH
        CASE ELSE           ' Weder ein WR noch ein ZV.
           ' Wenn das vorhergehende Zeichen ein WR
           ' war, dann setze die Variable VorDruRueck
           ' auf FALSCH und schreibe ein ZV vor das
           ' aktuelle Zeichen:
           IF VorDruRueck THEN
              VorDruRueck = FALSCH
              DateiZeich$ = ZeilenVor$ + DateiZeich$
           END IF
    END SELECT
    ' Schreibe das/die Zeichen in die neue Datei:
    PRINT #2, DateiZeich$;
LOOP
' Schreibe ein ZV, wenn das letzte Zeichen der
' Datei ein WR war:
IF VorDruRueck THEN PRINT #2, ZeilenVor$;
CLOSE                       ' Schließe beide Dateien.
PRINT "Eine weitere Datei (J/N)?"  ' Anfrage zum
                                   ' Fortführen.
' Verändere die Eingabe in Großbuchstaben:
Weiter$ = UCASE$(INPUT$(1))
```

```
' Setze das Programm fort, wenn der Benutzer ein "j"
' oder "J" eingegeben hat:
LOOP WHILE Weiter$ = "J"
END

FehlerBehand:           ' Fehlerbehandlungsroutine
   CONST KEINEDATEI = 53, DATEIDA = 58

   ' Die Funktion ERR gibt den Fehlercode des letzten
   ' Fehlers an:

   SELECT CASE ERR
      CASE KEINEDATEI  ' Programm konnte Datei mit dem
                       ' eingegebenen Namen nicht
                       ' finden.
         PRINT "Keine solche Datei im aktuellen "
         PRINT "Verzeichnis vorhanden."
         INPUT "Geben Sie neuen Namen ein: ",AusDatei$
         InDatei$ = Sicherung$(AusDatei$)
         RESUME
      CASE DATEIDA     ' Der Dateiname <Dateiname>.bak
                       ' existiert bereits in diesem
                       ' Verzeichnis: entferne die
                       ' Datei und fahre dann fort.
         KILL InDatei$
         RESUME
      CASE ELSE        ' Ein unerwarteter Fehler trat
                       ' auf: beende das Programm.
         ON ERROR GOTO 0
   END SELECT
'
' ===================== SICHERUNG$ =======================
'   Diese Prozedur gibt einen Dateinamen an, der aus
'   dem Basisnamen der eingegebenen Datei (alles vor
'   dem ".") plus der Erweiterung ".bak" besteht.
' ========================================================
'
FUNCTION Sicherung$ (DateiName$) STATIC
   ' Suche nach einem Punkt:
   Erweiterung = INSTR(DateiName$, ".")

   ' Wenn ein Punkt vorhanden ist, füge .bak zu der
   ' Basis hinzu:
   IF Erweiterung > 0 THEN
      Sicherung$ = LEFT$(DateiName$, Erweiterung-1) + ".bak"
```

```
   ' Andernfalls füge dem ganzen Namen .bak hinzu:
   ELSE
      Sicherung$ = DateiName$ + ".bak"
   END IF
END FUNCTION
```

2 Prozeduren: Unterprogramme und Funktionen

Dieses Kapitel beschreibt, wie Programme durch Aufteilen in kleinere logische Komponenten vereinfacht werden. Diese Komponenten mit dem Namen "Prozeduren" können dann Bausteine bilden, die die Sprache BASIC selbst verbessern und erweitern.

Dieses Kapitel erklärt auch, wie folgende Aufgaben anhand der Prozeduren zu bewältigen sind:

- Definition und Aufruf von BASIC-Prozeduren.
- Verwenden von lokalen und globalen Variablen in Prozeduren.
- Verwenden von Prozeduren anstelle von **GOSUB**-Unterroutinen und **DEF FN**-Funktionen.
- Übergabe von Argumenten an Prozeduren und Rückgabe von Werten aus Prozeduren.
- Schreiben rekursiver Prozeduren (Prozeduren, die sich selbst aufrufen können).

Obwohl sich ein BASIC-Programm anhand jedes beliebigen Texteditors erstellen läßt, erleichtert der QuickBASIC-Editor das Schreiben von Programmen mit Prozeduren erheblich. Zudem erzeugt QuickBASIC in den meisten Fällen beim Speichern des Programms automatisch eine **DECLARE**-Anweisung. (Eine **DECLARE**-Anweisung gewährleistet die Übergabe der korrekten Anzahl und Typen von Argumenten an eine Prozedur und erlaubt dem Programm, die in separaten Modulen definierten Prozeduren aufzurufen.)

2.1 Prozeduren: Bausteine für die Programmierung

In diesem Kapitel umfaßt der Begriff "Prozedur" sowohl die **SUB...END... SUB**- als auch die **FUNCTION...END FUNCTION**-Konstruktion. Prozeduren eignen sich zur Zusammenfassung sich wiederholender Aufgaben. Nehmen Sie beispielsweise an, daß Sie ein Programm schreiben, das Sie schließlich als selbständige Anwendung kompilieren möchten, und nehmen Sie weiter an, daß der Benutzer dieser Anwendung die Möglichkeit haben soll, verschiedene Argumente an diese Anwendung von der Befehlszeile aus zu übergeben. Dann ist es sinnvoll, diese Aufgabe – Aufteilung der von der Funktion **COMMAND$** angegebenen Zeichenkette in zwei oder mehrere Argumente – in eine separate Prozedur umzuwandeln. Ist diese Prozedur einmal erstellt und lauffähig, kann sie auch in anderen Programmen verwendet werden. Im Grund passen Sie BASIC durch die Verwendung von Prozeduren an Ihre eigenen Bedürfnisse an.

Die Programmierung anhand von Prozeduren hat folgende Vorteile:

1. Prozeduren ermöglichen das Aufteilen der Programme in einzelne logische Einheiten, die somit einfacher getestet werden können als ein gesamtes Programm ohne Prozeduren.
2. In einem Programm benutzte Prozeduren können normalerweise ohne bzw. mit nur kleinen Änderungen auch in anderen Programmen als Bausteine eingesetzt werden.

Prozeduren können ebenfalls in der eigenen Quick-Bibliothek abgelegt werden. Es handelt sich um eine besondere Datei, die sich beim Start von QuickBASIC in den Speicher laden läßt. Wenn sich erst einmal der Inhalt einer Quick-Bibliothek mit QuickBASIC im Speicher befindet, hat jedes geschriebene Programm Zugriff auf die Prozeduren in der Bibliothek. Dies vereinfacht es für jedes Programm, Programmcodes sowohl gemeinsam zu nutzen als auch zu sparen. (In Anhang H, "Erstellen und Verwenden von Quick-Bibliotheken" finden Sie weitere Informationen zum Erstellen von Quick-Bibliotheken.)

2.2 Prozeduren im Vergleich mit Unterroutinen und Funktionen

Wenn Sie frühere Versionen von BASIC kennen, können Sie sich eine **SUB...END SUB**-Prozedur ähnlich einer **GOSUB...RETURN**-Unterroutine vorstellen. Ebenso werden Sie einige Ähnlichkeiten zwischen einer **FUNCTION...END FUNCTION**-Prozedur und einer **DEF FN...END DEF**-Funktion feststellen. Wie in den Kapiteln 2.2.1 und 2.2.2 gezeigt, haben Prozeduren jedoch zahlreiche Vorteile gegenüber diesen älteren Konstruktionen.

Hinweis Um Verwechslungen zwischen **SUB**-Prozeduren und dem Ziel einer **GOSUB**-Anweisung zu vermeiden, werden in diesem Handbuch **SUB**-Prozeduren als "Unterprogramme" und Anweisungsblöcke, auf die der Zugriff durch **GOSUB...RETURN**-Anweisungen möglich ist, als "Unterroutinen" bezeichnet.

2.2.1 SUB und GOSUB im Vergleich

Obwohl die **GOSUB**-Unterroutine bei der Aufteilung eines Programms in überschaubare Einheiten behilflich ist, weisen **SUB**-Prozeduren eine Reihe von Vorteilen gegenüber Unterroutinen auf:

2.2.1.1 Lokale und globale Variablen

In **SUB**-Prozeduren sind alle Variablen standardmäßig lokale Variablen; das heißt, daß sie nur im Geltungsbereich der Definition der **SUB**-Prozedur existieren. Zur Verdeutlichung wird im folgenden Beispiel gezeigt, daß die Variable `I` in dem Unterprogramm `Test` lokal zu `Test` ist und mit der Variablen `I` im Modul-Ebenen-Code in keinem Zusammenhang steht:

```
I = 1
CALL Test
PRINT I    ' I ist immer noch gleich eins.
END

SUB Test STATIC
      I = 50
END SUB
```

Eine **GOSUB**-Unterroutine hat als Baustein im Programm einen erheblichen Nachteil: sie enthält nur "globale Variablen". Wenn sich eine Variable `I` innerhalb der Unterroutine und eine andere Variable namens `I` außerhalb der Unterroutine, aber im gleichen Modul, befinden, sind sie ein und dasselbe. Alle Änderungen des Wertes von `I` in der Unterroutine wirken sich auf `I` überall dort aus, wo es im Modul erscheint. Das Ergebnis ist, daß beim Versuch des Einbindens einer Unterroutine aus einem anderen Modul zur Vermeidung von Konflikten mit Variablennamen des neuen Moduls, die Variablen der Unterroutine eventuell umbenannt werden müssen.

2.2.1.2 Verwenden in Programmen mit mehreren Modulen

Eine **SUB** kann in einem Modul definiert und von einem anderen aus aufgerufen werden. Dies hat eine deutliche Verringerung des von einem Programm erforderten Codes, sowie eine Vereinfachung in der gemeinsamen Verwendung des Codes durch verschiedene Programme zur Folge.

Eine **GOSUB**-Unterroutine muß jedoch im selben Modul definiert und benutzt werden.

2.2.1.3 Arbeiten mit verschiedenen Sätzen von Variablen

Eine **SUB**-Prozedur kann beliebig oft innerhalb eines Programmes mit einem jeweils unterschiedlichen Satz von Variablen aufgerufen werden. Dieses Verfahren wird durch den Aufruf der **SUB** mit einer Argumentenliste durchgeführt. (Weitere Informationen hierzu finden Sie im Abschnitt 2.5, "Übergabe von Argumenten an Prozeduren"). Im nächsten Beispiel wird das Unterprogramm `Vergleich` zweimal mit jeweils unterschiedlichen Paaren von Variablen aufgerufen:

```
X = 4: Y = 5
CALL Vergleich (X, Y)
Z = 7: W = 2
CALL Vergleich (Z, W)
END

SUB Vergleich (A, B)
   IF A < B THEN SWAP A, B
END SUB
```

Es ist schwierig, eine **GOSUB**-Unterroutine mehr als einmal in demselben Programm aufzurufen, wenn diese jedesmal einen unterschiedlichen Satz von Variablen bearbeiten soll. Dieser Vorgang bringt das Kopieren von Werten in und aus globalen Variablen mit sich, wie im folgenden Beispiel verdeutlicht:

```
X = 4: Y = 5
A = X: B = Y
GOSUB Vergleich
X = A: Y = B

Z = 7 : W = 2
A = Z : B = W
GOSUB Vergleich
Z = A : W = B
END

Vergleich:
   IF A < B THEN SWAP A, B
RETURN
```

2.2.2 FUNCTION im Vergleich mit DEF FN

Während die Definition einer vielzeiligen **DEF FN**-Funktion der Forderung nach komplexeren Aufgaben, als auf einer Zeile untergebracht werden können, entspricht, bieten **FUNCTION**-Prozeduren dasselbe Leistungsvermögen zusätzlich zu den unten aufgeführten Vorteilen:

2.2.2.1 Lokale und globale Variablen

Standardmäßig sind innerhalb einer **FUNCTION**-Prozedur alle verwendeten Variablen zu ihr lokal, obwohl die Möglichkeit besteht, globale Variablen zu benutzen. (Weitere Informationen zu Prozeduren und globalen Variablen entnehmen Sie bitte Abschnitt 2.6, "Die gemeinsame Nutzung von Variablen durch SHARED".)

In einer **DEF FN**-Funktion sind Variablen innerhalb der Funktion standardmäßig global zu dem laufenden Modul (dies gilt ebenso für **GOSUB**-Unterroutinen). In einer **DEF FN**-Funktion kann jedoch eine Variable durch Schreiben in eine **STATIC**-Anweisung lokal gemacht werden.

2.2.2.2 Verändern von Variablen, die an Prozeduren übergeben werden

Variablen werden an **FUNCTION**-Prozeduren als Referenz oder als Wert übergeben. Wird eine Variable als Referenz übergeben, kann diese durch Änderung ihres zugehörigen Parameters in der **FUNCTION** verändert werden. Nach dem Aufruf `HoleRest` im nächsten Programm beträgt, zum Beispiel, der Wert von `X` zwei, da der Wert von `M` am Ende der **FUNCTION** zwei ist:

```
X = 89
Y = 40
PRINT HoleRest (X, Y)
PRINT X, Y                ' X ist jetzt 2.
END

FUNCTION HoleRest (M, N)
   HoleRest = M MOD N
   M = M \ N
END FUNCTION
```

Variablen werden an eine **DEF FN**-Funktion nur als Wert übergeben, so daß im nächsten Beispiel `FNRest M` verändert, ohne daß dabei `X` betroffen ist:

```
DEF FNRest (M, N)
   FNRest = M MOD N
   M = M \ N
END DEF
X = 89
Y = 40
PRINT FNRest (X, Y)
PRINT X,Y    ' X ist immer noch 89
```

(Weitere Informationen zum Unterschied zwischen Referenzübergabe und Wertübergabe finden Sie in den Abschnitten 2.5.5 und 2.5.6.)

2.2.2.3 Aufruf der Prozedur aus ihrer eigenen Definition heraus

Eine **FUNCTION**-Prozedur kann "rekursiv" sein, mit anderen Worten, kann sie sich aus ihrer eigenen Definition heraus aufrufen. (Weitere Informationen zu rekursiven Prozeduren sind Abschnitt 2.9 zu entnehmen.) Eine **DEF FN**-Funktion kann nicht rekursiv sein.

2.2.2.4 Verwendung in Programmen mit mehreren Modulen

Eine **FUNCTION**-Prozedur kann in einem Modul definiert und in einem anderen Modul verwendet werden. In dem Modul, in dem die **FUNCTION** verwendet wird, muß eine **DECLARE**-Anweisung eingesetzt werden; andernfalls nimmt das Programm an, daß sich der **FUNCTION**-Name auf eine Variable bezieht. (In Abschnitt 2.5.4, "Überprüfen von Argumenten anhand der **DECLARE**-Anweisung" sind weitere Informationen über die Benutzung der **DECLARE**-Anweisung zu diesem Zweck zu finden.)

Eine **DEF FN**-Funktion muß in demselben Modul definiert und verwendet werden. Zum Unterschied von **SUB**- oder **FUNCTION**-Prozeduren, die vor deren Erscheinen im Programm aufgerufen werden können, muß die **DEF FN**-Funktion immer vor der Verwendung in einem Modul definiert sein.

Hinweis Der Name einer **FUNCTION**-Prozedur kann jeder gültige BASIC-Variablenname sein, der nicht mit den Buchstaben **FN** beginnt. Dem Namen einer **DEF FN**-Funktion muß immer **FN** voranstehen.

2.3 Definieren der Prozeduren

BASIC-Prozedurdefinitionen weisen folgende allgemeine Syntax auf:

{**SUB** | **FUNCTION**} *Prozedurname* [(*Parameterliste*)] [**STATIC**]
  [*Anweisungsblock-1*]
  [**EXIT** {**SUB** | **FUNCTION**}
    [*Anweisungsblock-2*]]
END {**SUB** | **FUNCTION**}

Die folgende Liste beschreibt die Teile einer Prozedurdefinition:

Teil	*Beschreibung*
{**SUB** | **FUNCTION**}	Markiert den Anfang einer **SUB**- bzw. einer **FUNCTION**-Prozedur.
Prozedurname	Jeder gültige Variablenname mit bis zu 40 Zeichen. Derselbe Name kann nicht sowohl für eine **SUB** als auch für eine **FUNCTION** verwendet werden.
Parameterliste	Eine Liste von Variablen, die durch Kommata getrennt sind. Die Liste zeigt auch die Anzahl und Typen der an die Prozedur zu übergebenden Argumente an. (In Abschnitt 2.5.1 wird der Unterschied zwischen Parameter und Argument erläutert.)
STATIC	Wird das Attribut **STATIC** verwendet, sind lokale Variablen **STATIC**; das heißt, daß diese ihren Wert zwischen Prozeduraufrufen behalten. Wird das Attribut **STATIC** ausgelassen, sind lokale Variablen standardmäßig "automatisch", das bedeutet, daß diese am Anfang jedes Prozeduraufrufs zu Null oder zu Null-Zeichenketten initialisiert werden. Weitere Informationen entnehmen Sie bitte Abschnitt 2.7, "Automatische und STATIC -Variablen".

Fortsetzung auf der folgenden Seite.

Teil	*Beschreibung*
END {SUB | FUNCTION}	Beendet eine **SUB**- oder **FUNCTION** -Definition. Um korrekt zu funktionieren, muß eine Prozedur über genau eine **END{SUB | FUNCTION}**-Anweisung verfügen. Wenn das Programm eine **END SUB** oder **END FUNCTION** erreicht, verläßt es die Prozedur und springt zu der Anweisung zurück, die direkt auf diejenige folgt, die die Prozedur aufgerufen hat. Es können ebenso eine oder mehrere optionale **EXIT{SUB | FUNCTION}**-Anweisungen innerhalb der Prozedurdefinition zum Verlassen der Prozedur benutzt werden.

Innerhalb einer Prozedurdefinition sind alle gültigen BASIC-Ausdrücke und -Anweisungen erlaubt, mit Ausnahme der folgenden:

- **DEF FN...END DEF, FUNCTION...END FUNCTION** oder **SUB...END SUB**.

  Es ist also nicht möglich, Prozedurdefinitionen zu verschachteln oder eine **DEF FN**-Funktion innerhalb einer Prozedur zu definieren. Eine Prozedur kann jedoch eine andere Prozedur oder **DEF FN**-Funktion aufrufen.
- **COMMON**
- **DECLARE**
- **DIM SHARED**
- **OPTION BASE**
- **TYPE...END TYPE**

Beispiel

Das folgende Beispiel zeigt die **FUNCTION**-Prozedur mit dem Namen `GanzZahlPot`:

```
FUNCTION GanzZahlPot& (X&, Y&) STATIC
   PotenzWert& = 1
   FOR I& = 1 TO Y&
      Potenzwert& = PotenzWert& * X&
   NEXT I&
   GanzZahlPot& = PotenzWert&
END FUNCTION
```

2.4 Aufrufen von Prozeduren

In den nächsten beiden Abschnitten wird gezeigt, daß sich der Aufruf einer **FUNCTION**-Prozedur vom Aufruf einer **SUB**-Prozedur unterscheidet.

2.4.1 Aufrufen einer FUNCTION-Prozedur

Eine **FUNCTION**-Prozedur wird auf die gleiche Weise wie eine eingebaute BASIC-Funktion (z.B. **ABS**) durch Verwenden ihres Namens in einem Ausdruck aufgerufen.

```
' Jede der folgenden Anweisungen würde eine Funktion mit dem
' Namen "BisZehn" aufrufen :
PRINT 10 * BisZehn
X = BisZehn
IF BisZehn = 10 THEN PRINT "Außerhalb des Bereichs."
```

Eine **FUNCTION** kann Werte durch Änderung der ihr als Argumente übergebenen Variablen zurückgeben (eine Erläuterung zu diesem Verfahren finden Sie in Abschnitt 2.5.5, "Übergabe von Argumenten als Referenz"). Zusätzlich gibt eine **FUNCTION** in ihrem Namen einen einzigen Wert an, so daß der Name einer **FUNCTION** mit dem Typ, den sie zurückgibt, übereinstimmen muß. Wenn zum Beispiel eine **FUNCTION** einen Zeichenkettenwert angibt, muß entweder ihr Name ein angehängtes Dollar-Zeichen ($) haben, oder sie muß in einer vorhergehenden **DEFSTR**-Anweisung als zum Zeichenkettentyp gehörig deklariert sein.

Beispiel

Das folgende Programm stellt eine **FUNCTION** dar, die einen Zeichenkettenwert angibt. Beachten Sie, daß das Typdeklarationssuffix für Zeichenketten ($) zum Prozedurnamen gehört.

```
Eingabe$ = HolEingabe$     ' Rufe die FUNCTION auf und weise
                           ' den zurückgegebenen Wert einer
                           ' Zeichenkettenvariablen zu.
PRINT Eingabe$             ' Gib die Zeichenkette aus.
END
```

```
' ======================= HOLEINGABE$ =====================
'    Das Typdeklarationszeichen $ am Ende des Namens
'    dieser Funktion bedeutet, daß sie einen
'    Zeichenkettenwert angibt.
' =========================================================

FUNCTION HolEingabe$ STATIC
   ' Gib eine Zeichenkette mit 10 von der Tastatur gelesenen
   ' Zeichen an.
   ' Jedes Zeichen wird geschrieben, sobald es eingegeben
   ' ist:
   FOR I% = 1 TO 10
      Zeich$ = INPUT$ (1)      ' Hole das Zeichen.
      PRINT Zeich$;            ' Schreib das Zeichen auf den
                               ' Bildschirm.
      Temp$ = Temp$ + Zeich$   ' Füge das Zeichen der
                               ' Zeichenkette hinzu.

   NEXT
   PRINT
   HolEingabe$ = Temp$           ' Weise die Zeichenkette
                                 ' der FUNCTION zu.
END FUNCTION
```

Hinweis Obenaufgeführtes Programmbeispiel ist nur für die Verwendung in der QB-Umgebung bestimmt und läßt sich nicht anhand des BC-Befehls von DOS kompilieren.

2.4.2 Aufrufen einer SUB-Prozedur

Eine **SUB**-Prozedur unterscheidet sich von einer **FUNCTION**-Prozedur dadurch, daß sich eine **SUB** nicht durch Verwenden ihres Namens innerhalb eines Ausdruckes aufrufen läßt. Der Aufruf einer **SUB** ist eine Anweisung für sich, wie z.B. die BASIC-Anweisung **CIRCLE**. Ebenso gibt eine **SUB** einen Wert nicht mit ihrem Namen zurück, wie das bei einer **FUNCTION** der Fall ist. Eine **SUB** kann jedoch wie eine **FUNCTION** die Werte der an sie übergebenen Variablen verändern. (Eine Erläuterung zu diesem Verfahren finden Sie in Abschnitt 2.5.5, "Übergabe von Argumenten als Referenz".)

Eine **SUB** läßt sich durch die zwei folgenden Verfahren aufrufen:

1. Schreiben ihres Namens in eine **CALL**-Anweisung:

   ```
   CALL GibNachricht
   ```

2. Benutzen ihres Namens als eigene Anweisung:

   ```
   GibNachricht
   ```

Wird das Schlüsselwort **CALL** ausgelassen, dürfen die an die **SUB** übergebenen Argumente nicht in Klammern stehen:

```
' Rufe das Unterprogramm BearbEing mit CALL auf und übergib
' die drei Argumente Erstens$, Zweitens$ und AnzArg% an das
' Unterprogramm:
CALL BearbEing (Erstens$, Zweitens$, AnzArg%)
' Rufe das Unterprogramm BearbEing ohne CALL auf und übergib
' ihr die gleichen Argumente (Beachte: die Argumentenliste
' nicht in Klammern einschließen):
BearbEing Erstens$, Zweitens$, AnzArg%
```

Weitere Informationen zur Übergabe von Argumenten an Prozeduren finden Sie in Abschnitt 2.5.

Wenn das Programm **SUB**-Prozeduren nicht mit Hilfe von **CALL** aufruft und QuickBASIC nicht beim Erstellen des Programms verwendet wurde, muß der Name der **SUB** vor deren Aufruf in eine **DECLARE**-Anweisung eingesetzt werden.

```
DECLARE SUB TestAufTastatur
.
.
.
TestAufTastatur
```

Darauf müssen Sie nur achten, wenn Sie Programme außerhalb von QuickBASIC entwickeln, weil QuickBASIC beim Speichern von Programmen überall dort, wo sie benötigt werden, automatisch **DECLARE**-Anweisungen einfügt.

2.5 Übergabe von Argumenten an Prozeduren

Abschnitte 2.5.1 bis 2.5.4 erläutern, wie Parameter und Argumente unterschieden, Argumente an Prozeduren übergeben und Argumente überprüft werden, um sicherzustellen, daß sie die korrekte Anzahl und den korrekten Typ haben.

2.5.1 Parameter und Argumente

Der erste Schritt im Lernen der Übergabe von Argumenten an Prozeduren ist das Verstehen des Unterschiedes zwischen den Begriffen "Parameter" und "Argumente":

Parameter	***Argument***
Ein Variablenname, der in einer **SUB**-, **FUNCTION**- oder **DECLARE**-Anweisung erscheint.	Eine Konstante, Variable oder ein Ausdruck, die/der an eine **SUB** oder **FUNCTION** übergeben wird, wenn die **SUB** oder **FUNCTION** aufgerufen wird.

In einer Prozedurdefinition sind Parameter Platzhalter für Argumente. Wie in Abbildung 2.1 veranschaulicht, werden beim Aufruf einer Prozedur Argumente in die Variablen der Parameterliste übergeben, wobei der erste Parameter das erste Argument, der zweite Parameter das zweite Argument erhält, usw.

Abbildung 2.1 Parameter und Argumente

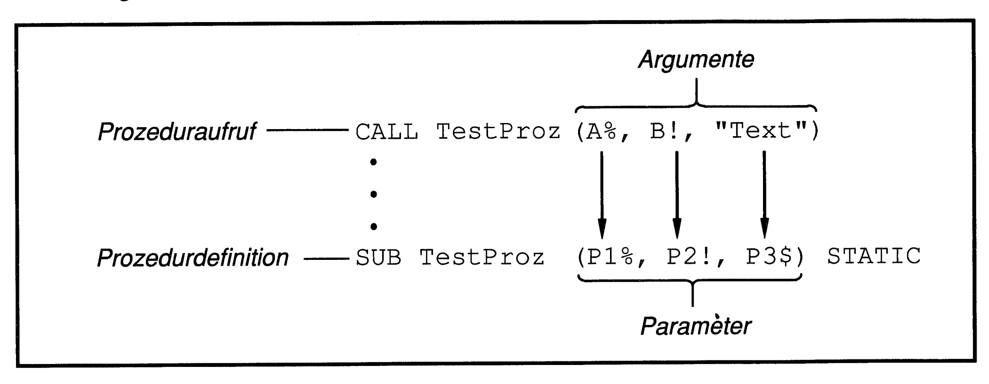


Abbildung 2.1 demonstriert ebenfalls eine weitere wichtige Regel: Obwohl die Namen von Variablen in einer Argumentenliste und einer Parameterliste nicht gleich sein müssen, muß die Anzahl von Parametern und Argumenten übereinstimmen. Darüberhinaus sollte der Typ (Zeichenkette, Ganzzahl, Zahl einfacher Genauigkeit, usw.) für zusammengehörende Argumente und Parameter gleich sein. (Weitere Informationen darüber, wie sichergestellt wird, daß Argumente und Parameter in Anzahl und Typ übereinstimmen, finden Sie in Abschnitt 2.5.4 "Überprüfen von Argumenten anhand der DECLARE-Anweisung".)

Eine Parameterliste besteht aus folgenden, durch Kommata getrennten Elementen:

- Gültigen Variablennamen, mit Ausnahme von Zeichenketten fester Länge

  `x$` und `x AS STRING` sind zum Beispiel beide gültig in einer Parameterliste, da sie sich auf Zeichenketten variabler Länge beziehen. Jedoch verweist `x AS STRING * 10` auf eine 10 Zeichen lange Zeichenkette fester Länge und kann nicht in einer Parameterliste erscheinen. (Zeichenketten fester Länge können aber als *Argumente* an Prozeduren übergeben werden. Weitere Informationen über Zeichenketten fester und variabler Länge sind Kapitel 4, "Zeichenkettenverarbeitung" zu entnehmen.)
- Datenfeldnamen, denen linke und rechte Klammern folgen

Eine Argumentenliste besteht aus folgenden, durch Kommata getrennten Elementen:

- Konstanten
- Ausdrücken
- gültigen Variablennamen
- Datenfeldnamen, denen linke und rechte Klammern folgen

Beispiele

Nachstehendes Beispiel zeigt die erste Zeile einer Unterprogrammdefinition mit einer Parameterliste:

```
SUB TestSub (A%, DatFeld(), DatsVar AS DatsTyp, Cs$)
```

Der erste Parameter, `A%`, ist eine Ganzzahl; der zweite Parameter, `DatFeld()`, ist ein Datenfeld einfacher Genauigkeit, da nichttypisierte Variablen standardmäßig von einfacher Genauigkeit sind; der dritte Parameter, `DatsVar`, ist ein Datensatz vom Typ `DatsTyp`; und der vierte Parameter, `Cs$`, ist eine Zeichenkette.

Die Zeile `CALL TestSub` im nächsten Beispiel ruft das Unterprogramm `TestSub` auf und übergibt dem Unterprogramm vier Argumente des passenden Typs:

```
TYPE DatsTyp
   Reihe     AS STRING * 12
   SerienNr  AS LONG
END TYPE

DIM DatsVar AS DatsTyp

CALL TestSub (X%, A(), DatsVar, "Gummibaum")
```

2.5.2 Übergabe von Konstanten und Ausdrücken

Konstanten - seien es Zeichenketten oder numerische Konstanten - können in der Liste der an die Prozedur übergebenen Argumente erscheinen. Selbstverständlich muß eine Zeichenkettenkonstante an einen Zeichenkettenparameter übergeben werden und eine numerische Konstante an einen numerischen Parameter, wie im nächsten Beispiel gezeigt:

```
CONST BILDBREITE = 80
CALL UebSchrift (BILDBREITE, "Monatlicher Bestandsbericht")
.
.
.
SUB UebSchrift (BD%, Titel$)
   .
   .
   .
END SUB
```

Wenn eine numerische Konstante in einer Argumentenliste nicht den gleichen Typ hat wie der zugehörige Parameter in der **SUB**- oder **FUNCTION**-Anweisung, wird die Konstante in den Typ des Parameters umgewandelt, wie aus der Ausgabe des nächsten Beispiels ersichtlich:

```
CALL Test (4.6, 4.1)
END

SUB Test (x%, y%)
   PRINT x%, y%
END SUB
```

Ausgabe

```
5     4
```

Aus Operationen mit Variablen und Konstanten resultierende Ausdrücke können ebenfalls an eine Prozedur übergeben werden. Genau wie bei Konstanten werden numerische Ausdrücke, die im Typ mit ihren Parametern nicht übereinstimmen, entsprechend umgewandelt, wie nachfolgend dargestellt:

```
Pruefe A! + 25!, NOT BooleWert%
' Im nächsten Aufruf wandelt die Angabe von Klammern die
' langganzzahlige Variable BWert& diese in einen Ausdruck
' um. Der Ausdruck (BWert&) wird in der SUB Pruefe in eine
' kurze Ganzzahl umgewandelt:
' Pruefe A!/3.1, (BWert&)
.
.
.
END
SUB Pruefe (Param1!, Param2%)
    .
    .
    .
END SUB
```

2.5.3 Übergabe von Variablen

Dieser Abschnitt beschreibt die Übergabe von einfachen Variablen, kompletten Datenfeldern, Elementen der Datenfelder, Datensätzen und Datensatzelementen an Prozeduren.

2.5.3.1 Übergabe von einfachen Variablen

Sowohl in einer Argumentenliste als auch in einer Parameterliste kann der Typ für eine einfache Variable auf eine der drei folgenden Arten deklariert werden:

1. Hinzufügen eines der folgenden Typdeklarationssuffixe zum Variablennamen: % & ! # oder $.
2. Deklarieren der Variablen in einer Klausel *Variablenname* **AS** *Typ*, wobei der *Typ* entweder **INTEGER**, **LONG**, **SINGLE**, **DOUBLE** oder **ZEICHENKETTEN** sein kann. Zum Beispiel:

   ```
   A AS LONG
   SUB Foo (A AS LONG)
   DECLARE SUB Foo (A AS LONG)
   ```

3. Zum Setzen des Standardtyps kann eine **DEF***Typ*-Anweisung verwendet werden.

Unabhängig von der gewählten Methode, müssen zusammengehörende Variablen denselben Typ sowohl in den Argumenten- als auch in den Parameterlisten aufweisen, wie aus nachstehendem Beispiel ersichtlich.

Beispiel

In diesem Beispiel werden zwei Argumente an die **FUNCTION**-Prozedur übergeben. Das erste ist eine Ganzzahl, die die Länge der von der **FUNCTION** angegebenen Zeichenkette anzeigt, während das zweite aus einem Zeichen besteht, das zur Bildung der Zeichenkette wiederholt wird.

```
FUNCTION ZeichKett$ (A AS INTEGER, B$) STATIC
   ZeichKett$ = STRING$(A%, B$)
END FUNCTION
DIM X AS INTEGER
INPUT "Geben Sie eine Zahl ein (1 bis 80): ", X
INPUT "Geben Sie ein Zeichen ein: ", Y$
' Gib eine Zeichenkette, die aus dem Zeichen Y$ besteht,
' X-mal aus:
PRINT ZeichKett$ (X, Y$)
END
```

Ausgabe

```
Geben Sie eine Zahl ein (1 bis 80): 21
Geben Sie ein Zeichen ein: #
#####################
```

2.5.3.2 Übergabe eines vollständigen Datenfeldes

Um alle Elemente eines Datenfeldes an eine Prozedur zu übergeben, ist der Name des Datenfeldes - gefolgt von linken und rechten Klammern - sowohl in die Argumentenliste als auch in die Parameterliste einzugeben.

Beispiel

Das folgende Beispiel zeigt, wie alle Elemente eines Datenfeldes an eine Prozedur übergeben werden:

```
DIM Werte (1 TO 5) AS INTEGER
' Beachte die leeren Klammern nach den Datenfeldnamen bei
' Prozeduraufruf und Übergabe der Datenfelder:
CALL WechsDatF (1, 5, Werte ())
CALL AusgDatF (1, 5, Werte ())
END
```

```
' Beachte die leeren Klammern nach dem Parameter P:
SUB WechsDatF (Min%, Max%, P() AS INTEGER) STATIC
   FOR I% = Min% TO Max%
      P (I%) = I% ^ 3
   NEXT I%
END SUB
SUB AusgDatF (Min%, Max%, P() AS INTEGER) STATIC
   FOR I% = Min% TO Max%
      PRINT P(I%),
   NEXT I%
   PRINT
END SUB
```

2.5.3.3 Übergabe von einzelnen Datenfeldelementen

Wenn eine Prozedur kein vollständiges Datenfeld erfordert, können stattdessen einzelne Elemente dieses Datenfeldes übergeben werden. Um ein Element eines Datenfeldes zu übergeben, ist der Datenfeldnamen, gefolgt von dem in Klammern eingeschlossenen passenden Index, zu verwenden.

Beispiel

Die Anweisung `WurzWert DatFeld (4, 2)` im nachstehenden Beispiel übergibt das Element in Zeile 4, Spalte 2 des Datenfeldes an das Unterprogramm `WurzWert`. (Beachten Sie, wie das Unterprogramm nun den Wert dieses Datenfeldelementes verändert):

```
DIM DatFeld (1 TO 5, 1 TO 3)
DatFeld(4, 2) = -36
PRINT DatFeld (4, 2)
WurzWert DatFeld (4, 2)
PRINT DatFeld (4, 2)   ' Der Aufruf von WurzWert hat den Wert
                       ' von DatFeld(4, 2) verändert.
END
SUB WurzWert(A) STATIC
   A = SQR(ABS(A))
END SUB
```

Ausgabe

```
-36
 6
```

2.5.3.4 Verwendung der Funktionen für Datenfeldgrenzen

Die Funktionen **LBOUND** und **UBOUND** sind bei der Bestimmung der Größe eines an eine Prozedur übergebenen Datenfeldes behilflich. Die Funktion **LBOUND** ermittelt den kleinsten Indexwert eines Datenfeldindexes, während die Funktion **UBOUND** den größten Indexwert ermittelt. Diese Funktionen ersparen Ihnen die Mühe, die untere und obere Grenze für jede Datenfelddimensionierung an eine Prozedur übergeben zu müssen.

Beispiel

Das Unterprogramm im folgenden Beispiel verwendet die **LBOUND**-Funktion bei der Initialisierung der Variablen `Zeile` und `Spalte` auf die jeweilig kleinsten Indexwerte der Dimensionen von A. Darüberhinaus verwendet es die Funktion **UBOUND**, um die Anzahl der Wiederholungen der **FOR**-Schleife auf die Anzahl der Elemente im Datenfeld zu begrenzen.

```
SUB Ausgabe(A(2)) STATIC
   FOR Zeile = LBOUND(A, 1) TO UBOUND(A, 1)
      FOR Spalte = LBOUND(A, 2) TO UBOUND(A, 2)
          PRINT A(Zeile, Spalte)
      NEXT Spalte
   NEXT Zeile
END SUB
```

2.5.3.5 Übergabe eines vollständigen Datensatzes

Zur Übergabe eines kompletten Datensatzes (eine als benutzerdefinierter Typ deklarierte Variable) an eine Prozedur ist wie folgt vorzugehen:

1. Definieren Sie den Typ (in diesem Beispiel `WarenBest`):

```
TYPE WarenBest
   TeilNum     AS STRING *  6
   Beschreib   AS STRING * 20
   EinzPreis   AS SINGLE
   Menge       AS INTEGER
END TYPE
```

2. Deklarieren Sie eine Variable (`WarenSatz`) mit diesem Typ:

```
DIM WarenSatz AS WarenBest
```

3. Rufen Sie eine Prozedur auf (`FindeSatz`) und übergeben Sie dieser die von Ihnen deklarierte Variable:

```
CALL FindeSatz (WarenSatz)
```

4. Geben Sie in der Prozedurdefinition dem Parameter den gleichen Typ wie der Variablen:

```
SUB FindeSatz (SatzVar AS WarenBest) STATIC
    .
    .
    .
END SUB
```

2.5.3.6 Übergabe einzelner Elemente eines Datensatzes

Um einzelne Elemente eines Datensatzes an eine Prozedur zu übergeben, ist der Name des Elementes (*Datensatzname.Elementname*) in die Argumentenliste einzusetzen. Vergewissern Sie sich, wie immer, daß der zugehörige Parameter in der Prozedurdefinition mit dem Typ dieses Elementes übereinstimmt.

Beispiel

Das folgende Beispiel zeigt, wie zwei Elemente der Datensatzvariablen `WarenSatz` an die **SUB**-Prozedur `SchrPreisEtik` übergeben werden. Beachten Sie, wie jeder Parameter in der **SUB**-Prozedur mit dem Typ der einzelnen Datensatzelementen übereinstimmt.

```
TYPE WarenBest
   TeilNum      AS STRING *  6
   Beschreib    AS STRING * 20
   EinzPreis    AS SINGLE
   Menge        AS INTEGER
END TYPE
DIM WarenSatz AS WarenBest
CALL SchrPreisEtik (WarenSatz.Beschreib,
WarenSatz.EinzPreis)
.
.
.
END
SUB SchrPreisEtik (Beschr$, Preis AS STRING)
    .
    .
    .
END SUB
```

2.5.4 Überprüfen von Argumenten anhand der DECLARE-Anweisung

Wenn Sie QuickBASIC zum Schreiben Ihrer Programme verwenden, werden Sie bemerken, daß QuickBASIC beim Speichern des Programmes für jede Prozedur automatisch eine **DECLARE**-Anweisung einfügt. Jede **DECLARE**-Anweisung besteht aus dem Wort **DECLARE**, gefolgt von den Worten **SUB** oder **FUNCTION**, dem Namen der Prozedur und einem Paar von Klammern. Die Klammern sind leer, wenn die Prozedur keine Parameter hat. Wenn die Prozedur Parameter hat, umschließen die Klammern eine *Parameterliste*, die die Anzahl und Typen der an die Prozedur zu übergebenden Argumente angibt. Diese *Parameterliste* hat dasselbe Format wie die Liste in der Definitionszeile der **SUB** oder **FUNCTION**.

Der Sinn der *Parameterliste* in einer **DECLARE**-Anweisung ist es, eine "Typüberprüfung" der an die Prozedur übergebenen Parameter einzuschalten. Das heißt, jedesmal wenn die Prozedur mit Variablenargumenten aufgerufen wird, werden diese Variablen überprüft, um sicherzustellen, daß sie mit der Anzahl und den Typen der Parameter in der **DECLARE**-Anweisung übereinstimmen.

Beim Speichern eines Programmes schreibt QuickBASIC alle Prozedurdefinitionen an das Ende eines Moduls. Daher würden Sie ohne das Vorhandensein von **DECLARE**-Anweisungen bei dem Versuch, dieses Programm mit dem Befehl BC zu kompilieren, auf das als "Vorwärtsbezug" (Aufruf einer Prozedur vor deren Definierung) bekannte Problem stoßen. Durch Erstellen eines Prototyps für die Prozedurdefinition ermöglicht die **DECLARE**-Anweisung dem Programm, Prozeduren aufzurufen, die erst später in diesem Modul oder in einem völlig anderen Modul definiert sind.

Beispiele

Das nächste Beispiel stellt eine **DECLARE**-Anweisung mit leerer Parameterliste dar, da keine Argumente an `HolEingabe$` übergeben werden:

```
DECLARE FUNCTION HolEingabe$ ()
X$ = HolEingabe$

FUNCTION HolEingabe$ STATIC
   HolEingabe$ = INPUT$(10)
END FUNCTION
```

Folgendes Beispiel weist eine Parameterliste in der **DECLARE**-Anweisung auf, da an diese Version von `HolEingabe$` ein Ganzzahlargument übergeben wird:

```
DECLARE FUNCTION HolEingabe$ (X%)
X$ = HolEingabe$ (5)

FUNCTION HolEingabe$ (X%) STATIC
   HolEingabe$ = INPUT$(X%)
END FUNCTION
```

2.5.4.1 Wann QuickBASIC keine DECLARE-Anweisung erzeugt

In einigen Fällen erzeugt QuickBASIC keine **DECLARE**-Anweisungen in dem Modul, das eine Prozedur aufruft.

QuickBASIC kann in einem Modul keine **DECLARE**-Anweisung für eine **FUNCTION**-Prozedur erzeugen, die in einem anderen Modul definiert ist, wenn das Modul nicht geladen ist.

In diesem Fall müssen Sie die **DECLARE**-Anweisung selbst an den Anfang des Moduls schreiben, in dem die **FUNCTION** aufgerufen wird; andernfalls betrachtet QuickBASIC den Aufruf der **FUNCTION** lediglich als einen Variablennamen.

QuickBASIC erzeugt keine **DECLARE**-Anweisung für eine **SUB**-Prozedur in einem anderen Modul, unabhängig davon, ob das Modul geladen ist oder nicht. Die **DECLARE**-Anweisung ist jedoch nur dann erforderlich, wenn Sie die **SUB**-Prozedur ohne das Schlüsselwort **CALL** aufrufen möchten.

Für eine Prozedur in einer Quick-Bibliothek kann QuickBASIC ebenfalls keine **DECLARE**-Anweisung erzeugen. In diesem Fall müssen Sie die Anweisung dem Programm selbst hinzufügen.

2.5.4.2 Entwicklung von Programmen außerhalb der QuickBASIC-Umgebung

Wenn Sie Ihre Programme mit dem eigenen Texteditor schreiben und anschließend mit den Befehlen BC und LINK außerhalb der QuickBASIC-Umgebung kompilieren, sind an den folgenden drei Stellen **DECLARE**-Anweisungen einzusetzen:

1. Am Anfang jedes Moduls, das eine **FUNCTION**-Prozedur vor deren Definierung aufruft:

```
DECLARE FUNCTION Hypot (X!, Y!)

INPUT X, Y
PRINT Hypot(X, Y)
END

FUNCTION Hypot (A, B) STATIC
   Hypot = SQR(A ^ 2 + B ^ 2)
END FUNCTION
```

2. Am Anfang jedes Moduls, das eine SUB-Prozedur vor deren Definierung aufruft und das kein **CALL** beim Aufruf der **SUB** verwendet:

```
DECLARE SUB AusgabeZeichK (X, Y)
INPUT X, Y

AusgabeZeichK X, Y     ' Beachte: keine Klammern um die
                       ' Argumente
END
```

```
SUB AusgabeZeichK(A,B) STATIC
   ' Übertrage die Zahlen in Zeichenketten, entferne alle
   ' führenden Leerzeichen aus der zweiten Zahl und gib
   ' aus :
   PRINT STR$(A) + LTRIM$(STR$(B))
END SUB
```

Wird eine **SUB**-Prozedur über **CALL** aufgerufen, ist es nicht erforderlich, die **SUB** zuerst zu deklarieren:

```
A$ = "466"
B$ = "123"
CALL AusgabeZeichK(A$, B$)
END

SUB AusgabeZeichK(X$, Y$) STATIC
   PRINT VAL(X$) + VAL(Y$)
END SUB
```

3. Am Anfang jedes Moduls, das eine in einem anderen Modul definierte **SUB**- oder **FUNCTION**-Prozedur aufruft (eine "externe Prozedur").

Wenn die Prozedur keine Parameter hat, vergessen Sie nicht, in der **DECLARE**-Anweisung nach dem Namen der Prozedur leere Klammern zu schreiben, wie im nächsten Beispiel gezeigt:

```
DECLARE FUNCTION LiesStunde$ ()
PRINT LiesStunde$
END

FUNCTION LiesStunde$ STATIC
   LiesStunde$ = LEFT$(TIME$,2)
END FUNCTION
```

Es ist zu beachten, daß eine **DECLARE**-Anweisung nur auf der Modul-Ebene, nicht auf der Prozedur-Ebene erscheinen kann. Eine **DECLARE**-Anweisung wirkt sich auf das gesamte Modul aus, in dem sie erscheint.

2.5.4.3 Verwendung von Include-Dateien für Deklarationen

Wurde ein separates Definitionsmodul erstellt, in dem eine oder mehrere **SUB**- oder **FUNCTION**-Prozeduren definiert sind, ist es empfehlenswert, eine entsprechende Include-Datei mit folgenden Bestandteilen anzulegen:

- **DECLARE**-Anweisungen für alle Modulprozeduren.
- **TYPE...END TYPE**-Verbunddefinitionen für alle Datensatzparameter der **SUB**- oder **FUNCTION**-Prozeduren in diesem Modul.
- **COMMON**-Anweisungen, die die von diesem Modul und anderen Modulen in dem Programm gemeinsam benutzten Variablen auflisten. (Weitere Informationen über die Verwendung von **COMMON** für diese Aufgabe sind in Abschnitt 2.6.3., "Die gemeinsame Nutzung von Variablen mit anderen Modulen" zu finden.)

Jedesmal, wenn Sie das Definitionsmodul in einem der Programme verwenden, fügen Sie zu Beginn jedes Moduls, das Prozeduren des Definitionsmoduls aufruft, einen **$INCLUDE**-Metabefehl ein. Ist das Programm kompiliert, wird der **$INCLUDE**-Metabefehl durch die aktuellen Inhalte der Include-Datei ersetzt.

Eine einfache Faustregel ist, für jedes Modul eine Include-Datei zu erstellen und anschließend das Modul und die Include-Datei, wie zuvor beschrieben, zusammen zu benutzen. Nachfolgende Liste führt einige Vorteile dieser Technik einzeln auf:

- Ein Modul, das Prozedurdefinitionen enthält, bleibt wirklich modular - das heißt, daß nicht alle **DECLARE**-Anweisungen für die Prozeduren des Moduls jedesmal bei deren Aufruf aus einem anderen Modul kopiert werden müssen; stattdessen genügt es, einen **$INCLUDE**-Metabefehl einzusetzen.
- In QuickBASIC unterdrückt die Benutzung einer Include-Datei für Prozedurdeklarationen die automatische Erzeugung von **DECLARE**-Anweisungen beim Speichern des Programms.
- Die Verwendung einer Include-Datei für Deklarationen vermeidet Probleme bei der Erkennung einer **FUNCTION** in einem Modul durch ein anderes Modul. (Weitere Informationen sind Abschnitt 2.5.4.1, "Wann QuickBASIC keine **DECLARE**-Anweisungen erzeugt" zu entnehmen.)

Die Einrichtung von QuickBASIC, **DECLARE**-Anweisungen zu erzeugen, kann auch beim Anlegen einer Include-Datei durch folgendes Verfahren benutzt werden:

1. Erstellen Sie das Modul.
2. Rufen Sie innerhalb dieses Moduls jede von Ihnen definierte **SUB**- oder **FUNCTION**-Prozedur auf.
3. Speichern Sie das Modul, um automatisch **DECLARE**-Anweisungen für alle Prozeduren zu erhalten.
4. Editieren Sie das Modul erneut, indem Sie alle Prozeduraufrufe entfernen und die **DECLARE**-Anweisungen in eine separate Include-Datei verschieben.

Im Anhang F, "Metabefehle" finden Sie weitere Informationen über die Syntax und die Verwendung von **$INCLUDE**-Metabefehlen.

Beispiel

Der folgende Abschnitt illustriert die gemeinsame Verwendung eines Definitionsmoduls und einer Include-Datei:

```
' ==========================================================
'                       MODDEF.BAS
' Dieses Modul enthält Definitionen für die Prozeduren
' ANFRAGE und MAX!.
' ==========================================================


FUNCTION Max! (X!, Y!) STATIC
   IF X! > Y! THEN Max! = X! ELSE Max! = Y!
END FUNCTION

SUB Anfrage (Zeile%, Spalte%, SatzVar AS SatzTyp) STATIC
   LOCATE Zeile%, Spalte%
   INPUT "Beschreibung: ", SatzVar.Beschreibung
   INPUT "Menge:        ", SatzVar.Menge
END SUB
' ==========================================================
'                       MODDEF.BI
' Dies ist eine Include-Datei, die DECLARE-Anweisungen
' enthält für die
' Prozeduren ANFRAGE und MAX! (genauso wie eine TYPE-
' Anweisung, die den Benutzertyp SatzTyp definiert).
' Verwenden Sie diese Datei immer dann, wenn
' Sie das Modul moddef.bas einsetzen.
' ==========================================================

TYPE SatzTyp
   Beschreibung AS STRING * 15
   Menge AS INTEGER
END TYPE

DECLARE FUNCTION Max! (X!, Y!)
DECLARE SUB Anfrage (Zeile%, Spalte%, SatzVar AS SatzTyp)
```

```
' ==========================================================
'                         BEISP.BAS
' Dieses Modul ist mit dem Modul moddef.bas Datensätzen und
' ruft die Prozeduren ANFRAGE und MAX! in moddef.bas auf.
' ==========================================================
' Die nächste Zeile läßt die Inhalte der Include-Datei
' moddef.bi ebenfalls Teil dieses Moduls werden:
' $INCLUDE: 'moddef.bi'

.
.
.
INPUT A, B
PRINT Max!(A, B)      ' Rufe die FUNCTION Max! in moddef.bas
                      ' auf.
.
.
.
Anfrage 5, 5, SatzVar ' Rufe die SUB Anfrage in moddef.bas
                      ' auf.
.
.
.
```

Wichtig Während es eine gute Programmierpraxis ist, Prozedurdeklarationen in eine Include-Datei zu schreiben, dürfen keine Prozedurdefinitionen (**SUB...END SUB** oder **FUNCTION...END FUNCTION**-Blöcke) in eine Include-Datei eingesetzt werden. Prozedurdefinitionen sind in QuickBASIC, Version 4.5 innerhalb von Include-Dateien nicht erlaubt. Wurden Include-Dateien zum Definieren von **SUB**-Prozeduren in Programmen verwendet, die mit den QuickBASIC-Versionen 2.0 oder 3.0 geschrieben sind, müssen diese Definitionen entweder in ein separates Modul geschrieben oder dort in das Modul eingefügt werden, wo sie aufgerufen werden.

2.5.4.4 Deklarieren von Prozeduren in Quick-Bibliotheken

Eine vorteilhafte Programmierpraxis ist es, alle Deklarationen von Prozeduren einer Quick-Bibliothek in eine Include-Datei zu schreiben. Mit dem Metabefehl **$INCLUDE** kann diese Include-Datei anschließend in Programme eingefügt werden, die die Bibliothek verwenden. Dadurch erübrigt sich das Kopieren aller erforderlichen **DECLARE**-Anweisungen bei der Benutzung der Bibliothek.

2.5.5 Übergabe von Argumenten als Referenz

Standardmäßig werden Variablen - ob einfache skalare Variablen, Datenfelder und Datenfeldelemente oder Verbunde - an **FUNCTION**- und **SUB**-Prozeduren "als Referenz" übergeben. Nachstehende Liste erläutert den Sinn der "Variablenübergabe als Referenz":

- Jede Programmvariable hat eine Adresse oder Stelle im Speicher, an der ihr Wert gespeichert ist.
- Der Aufruf einer Prozedur und die Übergabe von Variablen als Referenz an sie ruft die Prozedur auf und übergibt ihr die Adresse jeder Variablen. Daher sind die Adresse der Variablen und die Adresse des zugehörigen Parameters in der Prozedur ein und dasselbe.
- Deshalb ändert sich auch der Wert einer an sie übergebenen Variablen , wenn die Prozedur den Wert des Parameters verändert.

Ist die Wertänderung einer Variablen durch eine Prozedur unerwünscht, ist der in der Variablen enthaltene Wert, nicht die Adresse, zu übergeben. In diesem Fall werden Änderungen nur in einer Kopie der Variablen, nicht in der Variablen selbst, durchgeführt. Im nächsten Abschnitt sind weitere Erläuterungen zu dieser Möglichkeit zu finden.

Beispiel

Im folgenden Programm wird durch Ändern des Parameters `A$` in der Prozedur `Ersetze` auch das Argument `Test$` geändert:

```
Test$ = "eine Zeichenkette mit kleinbuchstaben."
PRINT "Vor Aufruf des Unterprogramms: "; Test$
CALL Ersetze (Test$, "t")
PRINT "Nach Aufruf des Unterprogramms: "; Test$
END

SUB Ersetze (A$, B$) STATIC
   Start = 1
   fDO
      ' Suche nach B$ in A$, beginnend am Zeichen mit
      ' Position Start in A$:
      Gefunden = INSTR(Start, A$, B$)

      ' Wandele jedes Vorkommen von B$ in A$ in einen
      ' Großbuchstaben um:
```

```
      IF Gefunden > 0 THEN
         MID$(A$,Gefunden) = UCASE$(B$)
         Start = Start + 1
      END IF
   LOOP WHILE Gefunden > 0
END SUB
```

Ausgabe

```
Vor Aufruf des Unterprogramms:
eine zeichenkette mit kleinbuchstaben.
Nach Aufruf des Unterprogramms:
eine ZeichenkeTTe mit kleinbuchsTaben
```

2.5.6 Übergabe von Argumenten als Wert

Die Übergabe eines Argumentes "als Wert" bedeutet, daß der Wert des Argumentes und nicht seine Adresse übergeben wird. In BASIC-Prozeduren wird die Übergabe einer Variablen als Wert simuliert, indem die Variable in eine temporäre Speicherstelle kopiert und dann die Adresse dieser temporären Speicherstelle übergeben wird. Da die Prozedur keinen Zugriff auf die Adresse der Ursprungsvariablen hat, kann sie die Ursprungsvariable nicht ändern. Stattdessen führt sie alle Änderungen mit der Kopie durch.

Ausdrücke können als Argumente an Prozeduren wie folgt übergeben werden:

```
' A + B ist ein Ausdruck; dieser Prozeduraufruf hat keine
' Auswirkungen auf die Werte von A und B:
CALL Mult(A + B, B)
```

Ausdrücke werden immer als Wert übergeben (weitere Informationen sind Abschnitt 2.5.2, "Übergabe von Konstanten und Ausdrücken" zu entnehmen).

Beispiel

Eine Möglichkeit zur Übergabe einer Variablen als Wert ist, diese in Klammern einzuschließen und somit in einen Ausdruck umzuwandeln. Wie aus nachstehender Ausgabe ersichtlich, werden Änderungen der Variablen `Y`, die zu der **SUB**-Prozedur lokal ist, als Änderungen der Variablen `B` an den Modul-Ebenen-Code zurückgegeben. Änderungen von `X` in der Prozedur haben jedoch keine Auswirkung auf den Wert von `A`, da `A` als Wert übergeben wird.

```
A = 1
B. = 1
PRINT "Vor Aufruf des Unterprogramms, A ="; A; ", B ="; B
' A wird als Wert übergeben, B wird als Referenz übergeben:
CALL Mult((A), B)
PRINT "Nach Aufruf des Unterprogramms, A ="; A; ", B ="; B
END
SUB Mult (X, Y) STATIC
   X = 2 * X
   Y = 3 * Y
   PRINT "Im Unterprogramm, X ="; X; ", Y ="; Y
END SUB
```

Ausgabe

```
Vor Aufruf des Unterprogramms, A = 1 , B = 1
Im Unterprogramm, X = 2 , Y = 3
Nach Aufruf des Unterprogramms, A = 1 , B = 3
```

2.6 Die gemeinsame Nutzung von Variablen anhand von SHARED

Zusätzlich zur Übergabe von Variablen durch Argument- und Parameterlisten können Prozeduren auf eine der zwei unten gezeigten Arten Variablen mit anderen Prozeduren und mit Code auf Modul-Ebene (der Code innerhalb eines Moduls, aber außerhalb jeglicher Prozedur) gemeinsam benutzen:

1. Innerhalb einer Prozedur in einer **SHARED**-Anweisung aufgeführte Variablen werden nur von dieser Prozedur und dem Modul-Ebenen-Code gemeinsam benutzt. Verwenden Sie diese Methode, wenn unterschiedliche Prozeduren im selben Modul verschiedene Kombinationen von Variablen der Modul-Ebene erfordern.
2. In einer **COMMON SHARED-**, **DIM SHARED-** oder **REDIM SHARED-**Anweisung auf Modul-Ebene aufgelistete Variablen werden von dem Modul-Ebenen-Code und allen Prozeduren innerhalb dieses Moduls gemeinsam benutzt. Diese Methode ist besonders vorteilhaft, wenn alle Prozeduren eines Moduls einen gemeinsamen Satz von Variablen verwenden.

Damit zwei oder mehrere Module Variablen gemeinsam benutzen, können auch die Anweisungen **COMMON** oder **COMMON SHARED** verwendet werden. Abschnitte 2.6.1 - 2.6.3 erläutern diese drei Möglichkeiten zur gemeinsamen Nutzung von Variablen.

2.6.1 Die gemeinsame Nutzung von Variablen mit bestimmten Prozeduren in einem Modul

Wenn unterschiedliche Prozeduren innerhalb eines Moduls unterschiedliche Variablen mit dem Modul-Ebenen-Code gemeinsam benutzen müssen, sollte die Anweisung **SHARED** in jeder Prozedur verwendet werden.

Datenfelder in **SHARED**-Anweisungen bestehen aus dem Namen des Datenfeldes gefolgt von leeren Klammern ():

```
SUB AndereSub STATIC
   SHARED FeldName ()
   .
   .
   .
```

Wird einer Variablen ihr Typ mit Hilfe einer **AS** *Typ*-Klausel zugewiesen, muß die **AS** *Typ*-Klausel zusammen mit der Variablen in einer **SHARED**-Anweisung erscheinen.

```
DIM Puffer AS STRING * 10
.
.
.
END
SUB LiesSatz STATIC
   SHARED Puffer AS STRING * 10
   .
   .
   .
END SUB
```

Beispiel

Im nächsten Beispiel weisen die **SHARED**-Anweisungen in den Prozeduren `HolSatz` und `InventGes` das Format einer Liste von gemeinsam benutzten Variablen auf:

```
' ============================================================
'                     MODUL-EBENEN-CODE
' ============================================================
TYPE SatzTyp
   Preis AS SINGLE
   Beschr AS STRING * 35
END TYPE
DIM SatzVar(1 TO 100) AS SatzTyp    ' Datensatzfeld
```

```
INPUT "Dateiname: ", DateiBeschr$
CALL HolSatz
PRINT InventGes
END
' ==========================================================
'                    PROZEDUR-EBENEN-CODE
' ==========================================================
SUB HolSatz STATIC
' Sowohl DateiBeschr$ als auch das Datensatzfeld SatzVar
' werden mit dem obigen Modul-Ebenen-Code gemeinsam benutzt:
   SHARED DateiBeschr$, SatzVar() AS SatzTyp
   OPEN DateiBeschr$ FOR RANDOM AS #1
    .
    .
    .
END SUB
FUNCTION InventGes STATIC
' Nur das Datenfeld SatzVar wird mit dem Modul-Ebenen-Code
' gemeinsam benutzt:
   SHARED SatzVar() AS SatzTyp
    .
    .
    .
END FUNCTION
```

2.6.2 Die gemeinsame Nutzung von Variablen mit allen Prozeduren in einem Modul

Wenn Variablen auf Modul-Ebene mit dem Attribut **SHARED** in einer **COMMON**-, **DIM**- oder **REDIM**-Anweisung deklariert sind (zum Beispiel der Ausdruck **COMMON SHARED** *Variablenliste*), dann haben alle Prozeduren innerhalb dieses Moduls Zugriff auf diese Variablen; mit anderen Worten macht das Attribut **SHARED** Variablen im gesamten Modul global.

Das Attribut **SHARED** ist bei der gemeinsamen Nutzung zahlreicher Variablen in allen Prozeduren eines Moduls hilfreich.

Beispiele

Nachfolgende Anweisungen deklarieren Variablen zur gemeinsamen Nutzung in allen Prozeduren eines einzelnen Moduls:

```
COMMON SHARED A, B, C
DIM SHARED Feld(1 TO 10, 1 TO 10) AS BenutzTyp
REDIM SHARED Alpha(N%)
```

Im folgenden Beispiel werden das Zeichenkettenfeld `ZeichFeld` und die Ganzzahlvariablen `Min` und `Max` vom Modul-Ebenen-Code und den beiden **SUB**-Prozeduren `FuelleFeld` und `SchreibFeld` gemeinsam benutzt:

```
' ============================================================
'                     MODUL-EBENEN-CODE
' ============================================================
DECLARE SUB FuelleFeld ()
DECLARE SUB SchreibFeld ()

' Wegen der folgenden DIM-Anweisungen werden
' die Ganzzahlvariablen Min und Max sowie das
' Zeichenkettendatenfeld ZeichFeld von allen
' SUB oder FUNCTION in diesem Modul gemeinsam benutzt:
DIM SHARED ZeichFeld (33 TO 126) AS STRING * 5
DIM SHARED Min AS INTEGER, Max AS INTEGER

Min = LBOUND(ZeichFeld)
Max = UBOUND(ZeichFeld)

FuelleFeld                ' Beachte die Abwesenheit von
                          ' Argumentenlisten.
SchreibFeld
END
' ============================================================
'                     PROZEDUR-EBENEN-CODE
' ============================================================
SUB FuelleFeld STATIC

   ' Belege jedes Element des Datenfeldes von 33 bis 126 mit
   ' einer Zeichenkette mit 5 Zeichen, von denen jedes
   ' Zeichen den ASCII-Code I% hat:
   FOR I% = Min TO Max
      ZeichFeld(I%) = STRING(5, I%)
   NEXT

END SUB

SUB SchreibFeld STATIC
   FOR I% = Min TO Max
      PRINT ZeichFeld(I%)
   NEXT
END SUB
```

Teilausgabe

```
!!!!!
"""""
#####
$$$$$
%%%%%
&&&&&
'''''
(((((
.
.
.
```

Wenn Sie zum Schreiben von Programmen einen eigenen Texteditor verwenden und diese Programme direkt außerhalb der QuickBASIC-Programmentwicklungsumgebung kompilieren, ist zu beachten, daß Variablendeklarationen mit dem Attribut **SHARED** vor der Prozedurdefinition stehen müssen. Andernfalls ist kein Wert der mit **SHARED** deklarierten Variablen in der Prozedur verfügbar, wie in der Ausgabe des nächsten Beispiels verdeutlicht. (Wird QuickBASIC zum Erstellen von Programmen verwendet, ist es nicht erforderlich, diesem Ablauf zu folgen, da QuickBASIC die Programme automatisch in der richtigen Reihenfolge speichert.)

```
DEFINT A-Z
FUNCTION Addiere (X, Y) STATIC
   Addiere = X + Y + Z
END FUNCTION
DIM SHARED Z
Z = 2
PRINT Addiere (1, 3)
END
```

Ausgabe

```
4
```

Das nächste Beispiel zeigt, wie das oben abgebildete Modul anhand der Anweisung `DIM SHARED Z` gefolgt von der Definition `Addiere` gespeichert wird:

```
DEFINT A - Z
DECLARE FUNCTION Addiere (X, Y)
' Die Variable Z wird nun mit Addiere gemeinsam benutzt:
```

```
DIM SHARED Z
Z = 2
PRINT Addiere (1, 3)
END

FUNCTION Addiere (X, Y) STATIC
   Addiere = X + Y + Z
END FUNCTION
```

Ausgabe

```
6
```

2.6.3 Die gemeinsame Nutzung von Variablen mit anderen Modulen

Wenn Sie im Programm Variablen zwischen Modulen gemeinsam benutzen möchten, listen Sie die Variablen in **COMMON**- oder **COMMON SHARED**-Anweisungen in jedem Modul auf der Modul-Ebene auf.

Beispiele

Das folgende Beispiel zeigt, wie Variablen zwischen Modulen durch Verwendung von **COMMON** in dem Modul, das die **SUB**-Prozedur aufruft, und **COMMON SHARED** in dem Modul, das die Prozedur definiert, gemeinsam benutzt werden. Mit **COMMON SHARED** haben alle Prozeduren des zweiten Moduls Zugriff auf die gemeinsamen Variablen:

```
' ============================================================
'                         HAUPT-MODUL
' ============================================================
COMMON A, B
A = 2,5
B = 1,2
CALL Quadrat
CALL Kubikzahl
END
' ============================================================
'       Modul mit den Prozeduren Kubikzahl und Quadrat
' ============================================================
' HINWEIS:  Die Namen der Variablen (X, Y) müssen nicht die
' gleichen sein wie in dem anderen Modul (A, B), müssen aber
' vom gleichen Typ sein.
```

```
COMMON SHARED X, Y   ' Diese Anweisung ist auf der Modul-
                     ' Ebene. Sowohl X als auch Y werden mit
                     ' den nachfolgenden Prozeduren KUBIKZAHL
                     ' und QUADRAT gemeinsam benutzt.

SUB Kubikzahl STATIC
   PRINT "A hoch 3  ="; X ^ 3
   PRINT "B hoch 3  ="; Y ^ 3
END SUB

SUB Quadrat STATIC
   PRINT "A quadrat ="; X ^ 2
   PRINT "B quadrat ="; Y ^ 2
END SUB
```

Das folgende Beispiel verwendet benannte **COMMON**-Blöcke auf den Modul-Ebenen und **SHARED**-Anweisungen innerhalb der Prozeduren, um unterschiedliche Sätze von Variablen mit jeder Prozedur gemeinsam zu benutzen:

```
' ==========================================================
'                        HAUPT-MODUL
' Gibt das Volumen und die Dichte eines gefüllten Zylinders
' mit den eingegebenen Werten aus
' ==========================================================

COMMON /VolWerte/ Hoehe, Radius, Volumen
COMMON /DichtWerte/ Gewicht, Dichte

INPUT "Höhe des Zylinders in Zentimeter: ", Hoehe
INPUT "Radius des Zylinders in Zentimeter: ", Radius
INPUT "Gewicht des gefüllten Zylinders in Gramm: ", Gewicht

CALL VolBerech
CALL DichteBerech

PRINT "Volumen ist"; Volumen; "Kubikzentimeter."
PRINT "Dichte ist"; Dichte; "Gramm/Kubikzentimeter."
END
' ==========================================================
'     Modul mit Prozeduren DichteBerech und VolBerech
' ==========================================================

COMMON /VolWerte/ H, R, V
COMMON /DichtWerte/ G, D

SUB VolBerech STATIC
```

```
   ' Benutze die Variablen Hoehe, Radius und Volumen
   ' gemeinsam mit dieser Prozedur:
   SHARED H, R, V
   CONST PI = 3,141592653589#
   V = PI * H (R ^ 2)
END SUB

SUB DichteBerech STATIC
   ' Benutze die Variablen Gewicht, Volumen und Dichte
   ' gemeinsam mit dieser Prozedur:
   SHARED G, V, D
   D = G / V
END SUB
```

Ausgabe

```
Höhe des Zylinders in Zentimeter: 100
Radius des Zylinders in Zentimeter: 10
Gewicht des gefüllten Zylinders in Gramm: 10000
Volumen ist 31415.93 Kubikzentimeter.
Dichte ist .3183099 Gramm/Kubikzentimeter.
```

2.6.4 Das Problem adreßgleicher Variablen

"Adreßgleiche Variablen" sind ein Problem, das manchmal in langen Programmen erscheint, die zahlreiche Variablen und Prozeduren enthalten. Adreßgleiche Variablen entstehen, wenn sich zwei oder mehrere Namen auf denselben Platz im Speicher beziehen. Dies kann in folgenden Situationen vorkommen:

- Wenn in der an eine Prozedur übergebenen Argumentenliste dieselbe Variable mehr als einmal erscheint.
- Wenn von einer Prozedur auf eine Variable, die in einer Argumentenliste übergeben wird, mit Hilfe der **SHARED**-Anweisung oder des **SHARED**-Attributes zugegriffen wird.

Zur Vermeidung der aus der Adreßgleichheit resultierenden Probleme sind mit einer Prozedur gemeinsam benutzte Variablen doppelt zu überprüfen, um sicherzustellen, daß diese Variablen nicht auch in der Argumentenliste eines Prozeduraufrufs erscheinen. Ebenso dürfen dieselben Variablen nicht zweimal, wie in der folgenden Anweisung, übergeben werden:

```
' X wird zweimal übergeben; dies wird in der Prozedur Test
' zu einem Problem adreßgleicher Variablen führen:
CALL Test (X, X, Y)
```

Beispiel

Das folgende Beispiel verdeutlicht das Auftreten von adreßgleichen Variablen. Hier wird die Variable `A` zwischen dem Modul-Ebenen-Code und der **SUB**-Prozedur gemeinsam mit der Anweisung `DIM SHARED` benutzt. `A` ist jedoch auch ein Argument, das als Referenz an die **SUB** übergeben wird. Daher beziehen sich in dem Unterprogramm sowohl `A` als auch `X` auf denselben Platz im Speicher. Demnach ändert das Unterprogramm auch `A`, wenn es `X` ändert, und umgekehrt.

```
DIM SHARED A
A = 4
CALL SchreibHaelfte(A)
END

SUB SchreibHaelfte (X) STATIC
   PRINT "Hälfte von"; X; "plus Hälfte von"; A; "ist
gleich";
   X = X / 2                ' X und A sind jetzt beide gleich
                            ' 2.
   A = A / 2                ' X und A sind jetzt beide gleich
                            ' 1.

   PRINT A + X
END SUB
```

Ausgabe

```
Hälfte von 4 plus Hälfte von 4 ist gleich 2
```

2.7 Automatische und STATIC-Variablen

Wenn das Attribut **STATIC** auf einer Prozedurdefinitionszeile erscheint, bedeutet dies, daß lokale Variablen innerhalb der Prozedur standardmäßig **STATIC** sind; das heißt, daß Werte zwischen den Aufrufen der Prozedur erhalten bleiben.

Das Auslassen des **STATIC**-Attributes läßt lokale Variablen innerhalb der Prozedur standardmäßig "automatisch" werden; das heißt, daß bei jedem Prozeduraufruf ein Satz lokaler Variablen mit neuen Werten erhalten wird.

Durch die Verwendung der **STATIC**-Anweisung innerhalb der Prozedur kann der Effekt, den das Auslassen des Attributes **STATIC** hinterläßt, überschrieben werden und somit einige Variablen automatisch und andere **STATIC** werden lassen (weitere Informationen sind in Abschnitt 2.8 zu finden).

Hinweis Die Anweisung **SHARED** überschreibt ebenfalls die Standardeinstellung für Variablen in einer Prozedur (lokal **STATIC** oder lokal automatisch), da jede Variable, die in einer **SHARED**-Anweisung erscheint, auf Modul-Ebene definiert und daher zu der Prozedur nicht lokal ist.

2.8 Beibehaltung der Werte von lokalen Variablen mit Hilfe der STATIC-Anweisung

Es kann vorkommen, daß Sie einige lokale Variablen in einer Prozedur **STATIC** machen möchten, während der Rest automatisch bleiben soll. Dies läßt sich durch die Aufführung dieser Variablen innerhalb der Prozedur in einer **STATIC**-Anweisung erreichen.

Um sicher zu gehen, daß die Variable lokal ist, kann ein Variablenname in eine **STATIC**-Anweisung geschrieben werden, da eine **STATIC**-Anweisung den Effekt einer **SHARED**-Anweisung auf Modul-Ebene überschreibt.

Eine **STATIC**-Anweisung kann nur innerhalb einer Prozedur erscheinen. Einem Datenfeldnamen in einer **STATIC**-Anweisung muß ein Satz leerer Klammern ( ) folgen. Ebenso muß jedes in einer **STATIC**-Anweisung vorkommende Datenfeld vor seiner Verwendung, wie im nächsten Beispiel gezeigt, dimensioniert werden:

```
SUB UnterProg2
   STATIC Datenfeld () AS INTEGER
   DIM Datenfeld(-5 TO 5, 1 TO 25) AS INTEGER
   .
   .
   .
END SUB
```

Hinweis Wird einer Variablen ihr Typ in einer Klausel **AS** Typ zugewiesen, muß die **AS**-Typ Klausel zusammen mit dem Namen der Variablen sowohl in der **STATIC** als auch in der **DIM**-Anweisung erscheinen.

Beispiel

Folgendes Beispiel veranschaulicht, wie eine **STATIC**-Anweisung den Wert der Zeichenkettenvariablen `Y$` während wiederholter Aufrufen von `TestSub` beibehält:

```
DECLARE SUB TestSub ()
FOR I% = 1 TO 5
   TestSub              ' Rufe TestSub fünfmal auf.
NEXT I%
END
```

```
SUB TestSub              ' Beachte: kein STATIC-Attribut
   ' Sowohl X$ als auch Y$ sind in TestSub lokale Variablen
   ' (das bedeutet, ihre Werte werden nicht gemeinsam mit
   ' dem Modul-Ebenen-Code benutzt). Da X$ jedoch eine
   ' automatische Variable ist, wird sie bei jedem Aufruf
   ' von TestSub zu einer Null-Zeichenkette initialisiert.
   ' Im Gegensatz dazu ist Y$ STATIC, so daß sie den Wert
   ' behält, den sie vom letzten Aufruf hat:
   STATIC Y$
   X$ = X$ + "*"
   Y$ = Y$ + "*"
   PRINT X$, Y$
END SUB
```

Ausgabe

```
*             *
*             **
*             ***
*             ****
*             *****
```

2.9 Rekursive Prozeduren

In BASIC können Prozeduren rekursiv sein. Eine rekursive Prozedur ist eine Prozedur, die sich selbst oder andere Prozeduren aufrufen kann, die die erste Prozedur erneut aufrufen.

2.9.1 Die Funktion Fakultät

Eine gute Möglichkeit, eine rekursive Prozedur zu verdeutlichen, ist die Betrachtung der Funktion Fakultät aus der Mathematik. Eine Möglichkeit, n! ("n-Fakultät") zu definieren, stellt die folgende Formel dar:

$n! = n*(n-1)*(n-2)*...*2*1$

Zum Beispiel wird 5-Fakultät wie folgt ausgewertet:

$5! = 5*4*3*2*1 = 120$

Hinweis Das in dieser Erläuterung verwendete mathematische Symbol für Fakultät darf nicht mit dem von QuickBASIC benutzten Typdeklarationssuffix für Zahlen einfacher Genauigkeit verwechselt werden.

Fakultäten eignen sich auch für eine rekursive Definition:

n! = n * (n-1)!

Dies führt zum folgenden Verlauf:

5! = 5*4!
4! = 4*3!
3! = 3*2!
2! = 2*1!
1! = 1*0!

Die Rekursion muß immer eine Abbruchbedingung aufweisen. Bei Fakultäten tritt diese Abbruchbedingung auf, wenn 0! ausgewertet wird, -per Definition ist 0! gleich 1.

Hinweis Obwohl eine rekursive Prozedur standardmäßig **STATIC**-Variablen enthalten kann (wie im nächsten Beispiel), ist es oft empfehlenswert, stattdessen automatische Variablen als Vorgabe zu verwenden. Auf dieser Weise wird die Möglichkeit ausgeschlossen, daß rekursive Aufrufe die Variablenwerte eines vorhergehenden Aufrufs überschreiben.

Beispiel

Nachstehendes Beispiel benutzt eine rekursive **FUNCTION**-Prozedur zur Berechnung von Fakultäten:

```
DECLARE FUNCTION Fakultaet# (N%)
Format$ = "###_! = ###################"
DO
   PRINT "Geben Sie eine Zahl von 0 bis 20 ein (oder -1 ";
   INPUT "zum Beenden): ", Zahl%
   IF Zahl% >= 0 AND Zahl% <= 20 THEN
      PRINT USING Format$; Zahl%; Fakultaet#(Zahl%)
   END IF
LOOP WHILE Zahl% >= 0
END

FUNCTION Fakultaet# (N%) STATIC
   IF N% > 0 THEN  ' Rufe Fakultaet# erneut auf, wenn N
                   ' größer als Null ist.
      Fakultaet# = N% * Fakultaet#(N% - 1)
   ELSE            ' Ist das Ende der rekursiven Aufrufe
                   ' erreicht (N% = 0), so "klettere die
                   ' Leiter wieder hinauf."
      Fakultaet# = 1
   END IF
END FUNCTION
```

2.9.2 Anpassen der Stapelgröße

Die Rekursion kann möglicherweise viel Speicherplatz einnehmen, da jeder Satz von automatischen Variablen in einer **SUB**- oder **FUNCTION**-Prozedur auf dem Stapel (Stack) gespeichert wird. (Das Speichern von Variablen auf diese Art erlaubt es einer Prozedur, mit korrekten Variablenwerten weiterzuarbeiten, nachdem die Kontrolle von einem rekursiven Aufruf zurückgegeben ist.)

Wenn es sich um eine rekursive Prozedur mit vielen automatischen Variablen oder um eine tief verschachtelte rekursive Prozedur handelt, muß eventuell die Größe des Stapels mit einer Anweisung **CLEAR**, *Stapelgröße* angepaßt werden, wobei *Stapelgröße* die Anzahl der zu reservierenden Bytes des Stapels bezeichnet. Andernfalls kann es vorkommen, daß Sie während des Programmlaufes die Fehlermeldung `Außerhalb des Stapelbereiches` erhalten.

Die folgenden Schritte zeigen eine Möglichkeit, die Größe des Speicherplatzes, den eine rekursive Prozedur benötigt, abzuschätzen:

1. Fügen Sie ein einzelnes Anführungszeichen ein, um den rekursiven Aufruf zeitweise in eine Kommentarzeile umzuwandeln, so daß die Prozedur während des Programmlaufes nur einmal aufgerufen wird.
2. Rufen Sie die Funktion **FRE(-2)**, die den gesamten unbenutzten Stapelplatz angibt, unmittelbar vor Aufruf der rekursiven Prozedur auf. Die Function **FRE(-2)** ist auch unmittelbar am Ende der rekursiven Prozedur aufzurufen. Lassen Sie die wiedergegebenen Werte anhand von **PRINT**-Anweisungen anzeigen.
3. Lassen Sie das Programm laufen. Der Wertunterschied besteht in der Größe des zum einmaligen Aufruf der Prozedur erforderlichen Stapelplatzes (in Bytes).
4. Schätzen Sie die Höchstanzahl der wahrscheinlichen Prozeduraufrufe und multiplizieren Sie anschließend diesen Wert mit dem von einem Prozeduraufruf eingenommenen Stapelplatz. Das Ergebnis wird als *Gesamtbytes* bezeichnet.
5. Reservieren Sie den im Schritt 4 berechneten Betrag an Stapelplatz: **CLEAR,,** *Gesamtbytes*

2.10 Übertragen der Kontrolle an ein anderes Programm anhand von CHAIN

Zum Unterschied von Prozeduraufrufen, die innerhalb desselben Programmes erscheinen, startet die Anweisung **CHAIN** einfach ein neues Programm. Wenn ein Programm mit einem anderen Programm verkettet wird, ergibt sich die folgende Reihenfolge:

1. Das erste Programm wird in seiner Ausführung angehalten.
2. Das zweite Programm wird in den Speicher geladen.
3. Das zweite Programm wird gestartet.

Der Vorteil der Verwendung von **CHAIN** besteht in der Aufteilung von Programmen mit großem Speicherbedarf in mehrere kleine Programme.

Die **COMMON**-Anweisung ermöglicht die Übergabe von Variablen eines Programmes an ein anderes in der Kette. Es ist übliche Programmierpraxis, diese **COMMON**-Anweisungen in eine Include-Datei zu schreiben und anschließend den Metabefehl **$INCLUDE** am Beginn jedes Programmes in der Kette zu verwenden.

Hinweis Es darf keine Anweisung **COMMON**/*Blockname*/*Variablenliste* (auch als "benannter **COMMON**-Block" bezeichnet) zur Übergabe von Variablen an ein verkettetes Programm verwendet werden, da die in einem **COMMON**-Block aufgeführten Variablen beim Verketten nicht erhalten bleiben. Es sind stattdessen unbenannte **COMMON**-Blöcke (**COMMON** *Variablenliste*) zu verwenden.

Beispiel

Dieses Beispiel, das aus einer Kette besteht, die drei einzelne Programme verbindet, verwendet eine Include-Datei, um Variablen zu deklarieren, die von den Programmen gemeinsam benutzt werden.

```
' =========== INHALT DER INCLUDE-DATEI COMMONS.BI ==========
DIM Werte(10)
COMMON Werte(), ZahlWerte
' ======================== HAUPT.BAS ======================
' Lies den Inhalt der COMMONS.BI-Datei ein:
' $INCLUDE: 'COMMONS.BI'
   ' Gib die Daten ein:
   PRINT "Geben Sie die Zahl der Datenwerte (<=10) ein: ";
   INPUT "", Zahlwerte
   FOR I = 1 TO ZahlWerte
      Anfrage$ = "Wert ("+LTRIM$(STR$(I))+")? "
      PRINT Anfrage$;
      INPUT "", Werte(I)
   NEXT I
   ' Der Benutzer soll die durchzuführende Berechnung
   ' angeben:
   PRINT "Berechnung(1=Standardabweichung,2=Mittelwert)?";
   INPUT "", Wahl
   ' Verkette nun zu dem richtigen Programm:
```

```
   SELECT CASE Wahl
      CASE 1:         ' Standardabweichung
         CHAIN "STABW"
      CASE 2:         ' Mittelwert
         CHAIN "MITTEL"
   END SELECT
END
' ========================STABW.BAS========================
' Berechnet die Standardabweichung eines Datensatzes
' ==========================================================
' $INCLUDE: 'COMMONS.BI'
   Sum     = 0   ' Normale Summe
   SumQuad = 0   ' Summe der quadrierten Werte
   FOR I = 1 TO ZahlWerte
      Sum     = Sum     + Werte(I)
      SumQuad = SumQuad + Werte(I) ^ 2
   NEXT I
   StdAbw = SQR(SumQuad / Zahlwerte - (Sum / Zahlwerte) ^ 2)
   PRINT "Die Standardabweichung der Stichprobe ist:";
   PRINT "StdAbw
END
' ======================= MITTEL.BAS ======================
' Berechnet den Mittelwert (Durchschnitt) eines Datensatzes
' ==========================================================
' $INCLUDE: 'COMMONS.BI'
   Sum = 0
   FOR I = 1 TO ZahlWerte
      Sum = Sum + Werte(I)
   NEXT
   Mittel = Sum / ZahlWerte
   PRINT "Der Mittelwert der Stichprobe ist: "Mittel
END
```

2.11 Anwendungsbeispiel: Rekursive Verzeichnisdurchsuchung (*woist.bas*)

Das folgende Programm verwendet eine rekursive **SUB**-Prozedur, `ScanDir`, um eine Festplatte nach dem Dateinamen zu durchsuchen, der vom Benutzer eingegeben wurde. Jedesmal, wenn dieses Programm die eingegebene Datei findet, gibt es den kompletten Pfad zu der Datei aus.

Verwendete Anweisungen und Funktionen

Dieses Programm demonstriert folgende in diesem Kapitel erläuterte Anweisungen und Schlüsselworte:

- **DECLARE**
- **FUNCTION...END FUNCTION**
- **STATIC**
- **SUB...END SUB**

Programm-Listing

```
DEFINT A-Z
' Deklariere die im Programm verwendeten symbolischen
' Konstanten:
CONST EOFTYP = 0, DATEITYP = 1, DIRTYP = 2, ROOT = "TWH"
DECLARE SUB SuchVerz (Pfad$, Ebene, DateiAng$,    Zeile)
DECLARE FUNCTION ErstDateiNam$ (Zahl)
DECLARE FUNCTION HolEintr$ (DateiZahl, EintrTyp)
CLS
INPUT "Zu suchende Datei"; DateiAng$
PRINT
PRINT "Geben Sie das Verzeichnis ein, ";
PRINT "ab dem gesucht werden soll"
PRINT "(optional Laufwerk + Verzeichnisse)."
PRINT "Betätigen Sie die Eingabetaste, um die Suche im ";
PRINT "Hauptverzeichnis des aktuellen Laufwerks zu "
PRINT "beginnen."
PRINT
INPUT "Startverzeichnis"; Pfad$
CLS
```

```
RechtsZeich$ = RIGHT$(Pfad$, 1)
IF Pfad$ = "" OR RechtsZeich$ = ":" OR _
   RechtsZeich$ <>   "\" THEN
   Pfad$ = Pfad$ + "\"
END IF
DateiAng$ = UCASE$(DateiAng$)
Pfad$ = UCASE$(Pfad$)
Ebene = 1
Zeile = 3
' Führe Aufruf mit oberster Ebene (Ebene 1) durch,
' um die Suche zu beginnen:
SuchVerz Pfad$, Ebene, DateiAng$, Zeile
KILL ROOT + ".*"    ' Lösche alle vom Programm
                    ' angelegten temporären Dateien.
LOCATE Zeile + 1, 1: PRINT "Suche beendet."
END
'
' ================== HOLEINTR =====================
'  Diese Prozedur verarbeitet Eintragszeilen einer
'  DIR-Liste, die in einer Datei gespeichert ist.
'
'  Diese Prozedur gibt die folgenden Werte aus:
'
'  HolEintr$    Ein gültiger Datei- oder
'               Verzeichnisname
'  EintrTyp     Wenn gleich 1, ist HolEintr$ eine
'               Datei
'               Wenn gleich 2, ist HolEintr$ ein
'               Verzeichnis.
'==================================================
'
FUNCTION HolEintr$ (DateiZahl, EintrTyp) STATIC
   ' Wiederhole, bis ein gültiger Eintrag oder Ende der
   ' Datei (EOF) gelesen ist:
   DO UNTIL EOF(DateiZahl)
      LINE INPUT #DateiZahl, EintrZeile$
      IF EintrZeile$ <> "" THEN
      ' Hole erstes Zeichen von der Zeile zur
      ' Überprüfung:
      TestZeich$ = LEFT$(EintrZeile$, 1)
     IF TestZeich$ <> " " AND TestZeich$ <> "." THEN EXIT DO
      END IF
   LOOP
```

```
   ' Entscheide, ob Eintrag oder EOF gefunden:
   IF EOF(DateiZahl) THEN      ' EOF, also gib EOFTYP
      EintrTyp = EOFTYP        ' in EintrTyp zurück.
      HolEintr$ = ""

   ELSE      ' Nicht EOF, also muß es eine Datei oder
             ' ein Verzeichnis sein.

     'Erstelle den Namen des Eintrags und gib ihn aus:
      EintrName$ = RTRIM$(LEFT$(EintrZeile$, 8))

     'Überprüfe auf Erweiterung und füge diese dem
     'Namen hinzu, sofern eine vorhanden ist:
      EintrErw$ = RTRIM$(MID$(EintrZeile$, 10, 3))
      IF EintrErw$ <> "" THEN
         HolEintr$ = EintrName$ + "." + EintrErw$
      ELSE
         HolEintr$ = EintrName$
      END IF

     'Bestimme den Typ des Eintrags und gib diesen
     'Wert zurück an die Stelle, an der HolEintr$
     'aufgerufen wurde.
      IF MID$(EintrZeile$, 15, 3) = "DIR" THEN
         EintrTyp = DIRTYP              ' Verzeichnis
      ELSE
         EintrTyp = DATEITYP            ' Datei
      END IF

   END IF
END FUNCTION
'
' ====================ERSTDATEINAM$=========================
'  Diese Prozedur erstellt einen Dateinamen aus
'  einer Basis-Zeichenkette ("TWH", als symbolische
'  Konstante auf Modul-Ebene definiert) und
'  einer an sie als Argument übergebenen Zahl (Zahl).
' =========================================================
'
FUNCTION ErstDateiNam$ (Zahl) STATIC
   ErstDateiNam$ = ROOT + "." + LTRIM$(STR$(Zahl))
END FUNCTION
'
```

```
' ====================== SUCHVERZ ==========================
' Diese Prozedur durchsucht rekursiv ein Verzeichnis
' nach dem Dateinamen, der vom Benutzer eingegeben
' wurde.
'
' BEACHTE: Der SUB-Kopf verwendet das Schlüsselwort
'          STATIC nicht, da diese Prozedur bei jedem
'          Aufruf einen neuen Satz von Variablen
'          benötigt.
' ==========================================================
'
SUB SuchVerz (Pfad$, Ebene, DateiAng$, Zeile)
   LOCATE 1, 1: PRINT "Ich durchsuche jetzt"; SPACE$(50);
   LOCATE 1, 15: PRINT Pfad$;
   ' Erstelle eine Dateibeschreibung für die temporäre
   ' Datei:
   TempBeschr$ = ErstDateiNam$(Ebene)
   ' Hole eine Verzeichnisliste des aktuellen
   ' Verzeichnisses und speichere sie in der
   ' temporären Datei:
   SHELL "DIR " + Pfad$ + " > " + TempBeschr$
   ' Hole die nächste verfügbare Dateinummer:
   DateiZahl = FREEFILE
   ' Öffne die DIR-Listendatei und durchsuche sie:
   OPEN TempBeschr$ FOR INPUT AS #DateiZahl
   ' Verarbeite die Datei Zeile für Zeile:
   DO
      'Lies den Eintrag aus der DIR-Listendatei ein:
      DirEintr$ = HolEintr$(DateiZahl, EintrTyp)
      ' Wenn Eintrag eine Datei ist:
      IF EintrTyp = DATEITYP THEN
      ' Wenn die Zeichenkette DateiAng$ passend ist,
      ' gib den Eintrag aus und verlasse diese
      ' Schleife:
      IF DirEintr$ = DateiAng$ THEN
         LOCATE Zeile, 1: PRINT Pfad$; DirEintr$;
         Zeile = Zeile + 1
         EintrTyp = EOFTYP
      END IF
```

```
      ' Wenn der Eintrag ein Verzeichnis ist, führe
      ' einen rekursiven Aufruf von SuchVerz mit dem
      ' neuen Verzeichnis durch:
      ELSEIF EintrTyp = DIRTYP THEN
         NeuPfad$ = Pfad$ + DirEintr$ + "\"
         SuchVerz NeuPfad$, Ebene + 1, DateiAng$, Zeile
         LOCATE 1,1:PRINT "Ich durchsuche jetzt";SPACE$(50);
         LOCATE 1, 15: PRINT Pfad$;
      END IF
   LOOP UNTIL EintrTyp = EOFTYP
   ' Die Suche in dieser Datei ist beendet, also
   ' schließe sie:
   CLOSE DateiZahl
END SUB
```

3 Datei- und Geräte-E/A

Dieses Kapitel beschreibt, wie BASIC Ein- und Ausgabe (E/A) Funktionen und Anweisungen zu verwenden sind, um einem Programm die Fähigkeit zu verleihen, auf in Dateien gespeicherte Daten zuzugreifen und Daten mit Geräten auszutauschen, die an das System angeschlossen sind.

Das Kapitel umfaßt vielfältige Programmieraufgaben, die sich auf das Wiederfinden, Speichern und Formatieren von Informationen beziehen. Dieses Thema behandelt auch die Beziehung zwischen Datendateien und physikalischen Geräten, wie z.B. Bildschirm und Tastatur.

Nach Abschluß dieses Kapitels werden Sie in der Lage sein, folgende Programmieraufgaben zu lösen:

- Ausgabe von Text auf den Bildschirm.
- Eingaben von der Tastatur zur Verwendung im Programm lesen.
- Datendateien auf Diskette anlegen.
- Speichern von Datensätzen in Datendateien.
- Lesen von Datensätzen aus Datendateien.
- Lesen oder Verändern von Daten in Dateien, die nicht auf dem American Standard Code for Information Interchange (ASCII) basieren.
- Daten mit anderen Computern über die serielle Schnittstelle austauschen.

3.1 Ausgabe von Text auf den Bildschirm

Dieser Abschnitt erläutert die Durchführung folgender Aufgaben:

- Ausgabe von Text auf den Bildschirm mit **PRINT**.
- Ausgabe von formatiertem Text auf den Bildschirm mit **PRINT USING**.
- Überspringen von Leerzeichen in einer Ausgabezeile mit **SPC**.
- Springen zu einer gegebenen Zeile in einer Ausgabezeile mit **TAB**.
- Verändern der Spalten- und Zeilenanzahl auf dem Bildschirm mit **WIDTH**.
- Öffnen eines Text-Darstellungsfeldes mit **VIEW PRINT**.

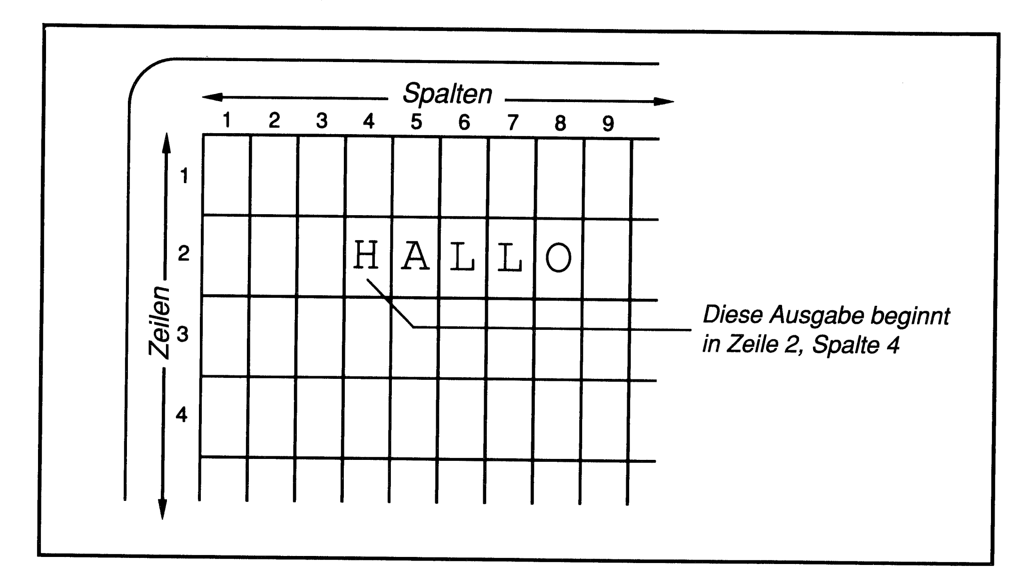
Hinweis Eine Ausgabe, die auf dem Bildschirm erscheint, wird manchmal als "Standardausgabe" bezeichnet. Sie können die Standardausgabe mit Hilfe der Symbole > oder >> auf der DOS-Befehlszeile umleiten, so daß für den Bildschirm bestimmte Ausgaben zu einem anderen Ausgabegerät (z.B. einem Drucker) oder zu einer Diskettendatei gesendet werden. (Nähere Informationen zur Umleitung der Eingabe finden Sie in der DOS-Dokumentation.)

3.1.1 Bildschirmzeilen- und spalten

Um die Ausgabe von Text auf den Bildschirm zu verstehen, ist es hilfreich, sich den Bildschirm als ein Raster von "Zeilen" und "Spalten" vorzustellen. Die Höhe einer Zeile überschreitet leicht die Höhe einer geschriebenen Zeile, während die Breite einer Spalte etwas größer als die Breite eines Zeichens ist. Eine Standard-Bildschirmkonfiguration im Textmodus (nicht graphisch) hat eine Breite von 80 Spalten bei einer Höhe von 25 Zeilen. Abbildung 3.1 zeigt, wie jedes auf den Bildschirm ausgegebene Zeichen ein einziges Element des Rasters einnimmt, das durch die Verbindung eines Zeilenarguments mit einem Spaltenargument festgelegt werden kann.

Die unterste Zeile des Bildschirms wird normalerweise nicht für Ausgaben verwendet, solange keine **LOCATE**-Anweisung benutzt wird, um dort Text auszugeben. (Weitere Informationen zu **LOCATE** sind Abschnitt 3.3, "Steuerung des Text-Cursors" zu entnehmen.)

Abbildung 3.1 Textausgabe auf den Bildschirm



3.1.2 Anzeige von Text und Zahlen anhand von PRINT

Die bei weitem am häufigsten verwendete Anweisung für Ausgaben auf den Bildschirm ist die Anweisung **PRINT**. Mit Hilfe von **PRINT** können numerische oder Zeichenkettenwerte, sowie eine Kombination aus beiden angezeigt werden. Außerdem gibt **PRINT** ohne Argumente eine leere Zeile aus.

Es folgen einige allgemeine Erläuterungen zu **PRINT**:

- **PRINT** gibt Zahlen immer mit einem nachfolgenden Leerzeichen aus. Wenn die Zahl positiv ist, geht ihr auch immer ein Leerzeichen voraus; wenn die Zahl negativ ist, geht ihr ein Minuszeichen voraus.
- **PRINT** kann zur Ausgabe von Ausdruckslisten verwendet werden. Ausdrücke in einer Liste können durch Kommata, Semikolons, ein oder mehrere Leerzeichen, sowie durch ein oder mehrere Tabulatorzeichen von anderen Ausdrücken getrennt werden. Ein Komma veranlaßt **PRINT**, auf dem Bildschirm zum Anfang der nächsten "Ausgabezone", also zum Anfang des nächsten 14-spaltigen Blocks zu springen. Ein Semikolon (oder eine Kombination von Leerzeichen und/oder Tabs) zwischen zwei Ausdrücken schreibt die Ausdrücke auf den Bildschirm direkt nebeneinander ohne Leerzeichen (abgesehen von den "eingebauten" Leerzeichen für Zahlen).
- Normalerweise beendet **PRINT** jede Ausgabezeile mit einer Neue-Zeile (Wagenrücklauf-Zeilenvorschub) Sequenz. Ein Komma oder ein Semikolon am Ende der Ausdrucksliste unterdrückt dies jedoch; solange die nächste geschriebene Ausgabe des Programms nicht zu lang ist, um auf diese Zeile zu passen, erscheint sie auf derselben Zeile.

- **PRINT** bricht eine Ausgabezeile, die länger als die Breite des Bildschirms ist, auf die nächste Zeile um. Wenn Sie zum Beispiel versuchen, eine 100 Zeichen lange Zeile auf einen Bildschirm mit 80 Spalten auszugeben, werden die ersten 80 Zeichen der Zeile in eine Zeile geschrieben, gefolgt von den nächsten 20 Zeichen auf der nächsten Zeile. Beginnt die Zeile mit 100 Zeichen nicht an der linken Kante des Bildschirms (wenn sie, zum Beispiel, nach einem Komma oder nach einer mit einem Semikolon beendeten **PRINT**-Anweisung steht), wird die Zeile bis zur 80. Spalte einer Zeile geschrieben und in der ersten Spalte der nächsten Zeile fortgesetzt.

Beispiel

Die Ausgabe des folgenden Programmes stellt einige der unterschiedlichen Möglichkeiten der Verwendung von **PRINT** dar:

```
A = 2
B =-1
C = 3
X$ = "da"
Y$ = "hinten"
PRINT A, B, C
PRINT B, A, C
PRINT A; B; C
PRINT X$; Y$
PRINT X$, Y$;
PRINT A, B
PRINT
FOR I = 1 TO 8
   PRINT X$,
NEXT
```

Ausgabe

```
 2     -1     3
-1      2     3
 2     -1     3
dahinten
da     hinten 2     -1
da     da     da     da     da
da     da     da
```

3.1.3 Anzeige von formatierten Ausgaben anhand von PRINT USING

Zum Unterschied von **PRINT** ermöglicht die **PRINT USING**-Anweisung eine bessere Kontrolle über das Erscheinungsbild geschriebener Daten, insbesondere numerischer Daten. Mit Hilfe besonderer, in einer Formatzeichenkette eingebundener Zeichen, erlaubt **PRINT USING** die Angabe von Informationen, wie z.B. die Anzahl der anzuzeigenden Dezimalstellen einer Zahl oder der Zeichen einer Zeichenkette, sowie das Versehen einer Zahl mit einem Pluszeichen (+) oder einem Dollarzeichen ($), u.s.w.

Beispiel

Das folgende Beispiel zeigt verschiedene Anwendungen von **PRINT USING**. Es ist zu beachten, daß nach der Formatzeichenkette **PRINT USING** mehr als ein Ausdruck stehen kann. Wie in dem Fall von **PRINT**, können die Anweisungen in der Liste durch Kommata, Semikolons, Leerzeichen oder Tabzeichen getrennt sein.

```
X = 441.2318

PRINT USING "Die Zahl mit drei Dezimalstellen ###.###";X
PRINT USING "Die Zahl mit einem Dollarzeichen $$##.##";X
PRINT USING "Die Zahl im Exponentialformat #.###^^^^;X
PRINT USING "Zahlen mit Pluszeichen +###  "; X; 99.9
```

Ausgabe

```
Die Zahl mit drei Dezimalstellen 441.232
Die Zahl mit einem Dollarzeichen $441.23
Die Zahl im Exponentialformat 0.441E+03
Zahlen mit Pluszeichen +441  Zahlen mit Pluszeichen +100
```

Weitere Informationen über **PRINT USING** entnehmen Sie bitte dem QB Ratgeber.

3.1.4 Überspringen von Leerzeichen und Springen in eine angegebene Spalte

Mit Hilfe der Anweisung **SPC(*n*)** in einer **PRINT**-Anweisung lassen sich *n* Leerzeichen in einer Ausgabezeile überspringen, wie aus der Ausgabe des nächsten Beispiels ersichtlich:

```
PRINT "         1         2         3"
PRINT "123456789012345678901234567890"
PRINT "Vorname";  SPC(10);  "Nachname"
```

Ausgabe

```
1         2         3
123456789012345678901234567890
Vorname             Nachname
```

Mit Hilfe der **TAB(***n***)**-Anweisung in einer **PRINT**-Anweisung können Sie in die *n*-te Spalte (von der linken Seite des Bildschirms aus) einer Zeile der geschriebenen Ausgabe springen. Das folgende Beispiel erstellt dieselbe Ausgabe wie die des oberen Beispiels anhand von **TAB**:

```
PRINT "         1         2         3"
PRINT "123456789012345678901234567890"
PRINT "Vorname";  TAB(21);  "Nachname"
```

Weder **SPC** noch **TAB** können selbständig zur Positionierung von geschriebenen Ausgaben auf dem Bildschirm verwendet werden; sie können nur in **PRINT**-Anweisungen erscheinen.

3.1.5 Ändern der Zeilen- und Spaltenzahl

Die maximale Anzahl von Zeichen, die in einer einzigen Ausgabezeile erscheinen, kann anhand der Anweisung **WIDTH** *Spalten* gesteuert werden. Die Anweisung **WIDTH** *Spalten* verändert die Größe der auf den Bildschirm ausgegebenen Zeichen, so daß mehr bzw. weniger Zeichen in eine Zeile passen. **WIDTH 40**, zum Beispiel, verbreitert die Zeichen, so daß die maximale Zeilenlänge 40 Zeichen beträgt. **WIDTH 80** verkleinert Zeichen, so daß die maximale Zeilenlänge 80 Zeichen beträgt. Die Zahlen 40 und 80 sind die einzigen gültigen Werte für das Argument *Spalten*.

Auf Maschinen, die mit einem "Enhanced Graphics Adapter" (EGA) oder "Video Graphics Adapter" (VGA) ausgerüstet sind, kann die Anweisung **WIDTH** auch die Anzahl der auf dem Bildschirm erscheinenden Zeilen steuern, wie aus folgender Syntax ersichtlich:

WIDTH [*Spalten*][, *Zeilen*]

Der Wert für *Zeilen* kann 25, 30, 43, 50 oder 60 sein, je nach dem verwendeten Bildschirmadapter und dem in einer vorherigen **SCREEN**-Anweisung gesetzten Bildschirmmodus.

3.1.6 Erstellen eines Text-Darstellungsfeldes

Bisher wurde der gesamte Bildschirm für Textausgaben verwendet. Mit der Anweisung **VIEW PRINT** läßt sich jedoch die geschriebene Ausgabe auf ein "Text-Darstellungsfeld" begrenzen, das einen horizontalen Ausschnitt des Bildschirms bezeichnet. Die Anweisung **VIEW PRINT** weist folgende Syntax auf:

VIEW PRINT [*oberste Zeile* **TO** *unterste Zeile*]

Die Werte der *obersten* und *untersten Zeile* geben den Anfang und das Ende des Darstellungsfeldes an.

Ein Text-Darstellungsfeld gewährleistet ebenfalls die Kontrolle über das Rollen (scrolling) auf dem Bildschirm. Ohne ein Text-Darstellungsfeld beginnt die geschriebene Ausgabe zu rollen, wenn sie das untere Ende des Bildschirms erreicht hat. Sowohl Text, als auch Graphik, die sich auf der jeweilig obersten Zeile des Bildschirms befinden, gehen beim Rollen des Bildschirms verloren. Nach einer **VIEW PRINT**-Anweisung jedoch, findet Rollen nur noch zwischen der obersten und der untersten Zeile des Darstellungsfeldes statt. Das bedeutet, daß die angezeigte Ausgabe am oberen und/oder unteren Ende des Bildschirms gekennzeichnet werden kann, ohne daß die Kennzeichnung das Rollen der Zeilen herbeiführt, wenn zu viele Zeilen erscheinen. Um lediglich das Text-Darstellungsfeld zu löschen, kann zusätzlich die Anweisung **CLS 2** benutzt werden, wobei der Rest des Bildschirminhaltes erhalten bleibt. Das Erstellen eines graphischen Darstellungsfeldes für graphische Ausgaben auf dem Bildschirm wird in Abschnitt 5.5, "Definieren eines graphischen Darstellungsfeldes" beschrieben.

Beispiel

Die Auswirkungen einer **VIEW PRINT-**Anweisung lassen sich anhand der Ausgabe des nächsten Beispiels feststellen:

```
CLS
LOCATE 3, 1
PRINT "Oberhalb des Text-Darstellungsfeldes, ";
PRINT "es wird aber nicht gerollt."
LOCATE 4, 1
PRINT STRING$(60, "_")       ' Schreibe horizontale
LOCATE 11, 1                 ' Linie oberhalb und
PRINT STRING$(60, "_")       ' unterhalb des Text-
                             ' Darstellungsfeldes.
PRINT "Unterhalb des Text-Darstellungsfeldes."
VIEW PRINT 5 TO 10           ' Text-Darstellungsfeld erstreckt
                             ' sich von Zeile 5 bis 10.
FOR I = 1 TO 20              ' Schreibe Zahlen und Text in das
                             ' Darstellungsfeld.
   PRINT I; "eine Textzeile"
NEXT
DO: LOOP WHILE INKEY$ = ""   ' Warte, bis eine Taste
                             '  betätigt wird.
CLS 2                        ' Lösche nur das Darstellungsfeld.
END
```

Ausgabe (bevor der Benutzer eine Taste betätigt)

```
Oberhalb des Text-Darstellungsfeldes, es wird aber nicht gerollt.
_______________________________________________________________
 16 eine Textzeile
 17 eine Textzeile
 18 eine Textzeile
 19 eine Textzeile
 20 eine Textzeile

_______________________________________________________________
Unterhalb des Text-Darstellungsfeldes.
```

Ausgabe (nachdem der Benutzer eine Taste betätigt hat)

```
Oberhalb des Text-Darstellungsfeldes, es wird aber nicht gerollt.
_______________________________________________________________




_______________________________________________________________
Unterhalb des Text-Darstellungsfeldes.
```

3.2 Einlesen der Eingaben von der Tastatur

Dieser Abschnitt erklärt die Verwendung folgender Anweisungen und Funktionen, die BASIC-Programme in die Lage versetzen, per Tastatur eingegebene Eingaben einzulesen:

- **INPUT**
- **LINE INPUT**
- **INPUT$**
- **INKEY$**

Hinweis Eingaben über die Tastatur werden häufig als "Standardeingabe" bezeichnet. Sie können unter Verwendung des DOS-Symbols < bestimmen, ob die Standardeingabe für das Programm aus einer Datei oder einem anderen Eingabegerät, anstelle der Tastatur, kommen soll. (Weitere Informationen zur Datenumleitung finden Sie in der DOS-Dokumentation.)

3.2.1 Die INPUT-Anweisung

Die **INPUT**-Anweisung übernimmt vom Benutzer eingegebene Informationen und speichert diese wie folgt in einer Liste von Variablen:

```
INPUT A%, B, C$
INPUT D$
PRINT A%, B, C$, D$
```

Ausgabe

```
? 6.6,45,eine Zeichenkette
? "zwei, drei"
  7        45        eine Zeichenkette        zwei, drei
```

Es folgen einige allgemeine Erläuterungen zu **INPUT**:

- **INPUT** selbst fordert den Benutzer zu einer Eingabe mit einem Fragezeichen (?), gefolgt von einem blinkenden Cursor, auf.
- Auf die Anweisung **INPUT** folgen ein oder mehrere Variablennamen. Ist mehr als eine Variable vorhanden, werden diese durch Kommata voneinander getrennt.
- Die Anzahl von Konstanten, die vom Benutzer nach der **INPUT**-Eingabeaufforderung eingegeben werden, muß mit der Anzahl der Variablen in der **INPUT**-Anweisung selbst übereinstimmen.
- Die vom Benutzer eingegebenen Werte müssen im Typ mit den Variablen in der auf **INPUT** folgenden Liste übereinstimmen. Mit anderen Worten, ist eine Zahl einzugeben, wenn die Variable für den Typ Ganzzahl, lange Ganzzahl, einfache Genauigkeit oder doppelte Genauigkeit vorgesehen ist. Es ist dagegen eine Zeichenkette einzugeben, wenn die Variable für den Typ Zeichenkette vorgesehen ist.
- Da Konstanten in einer Eingabeliste durch Kommata voneinander getrennt sein müssen, sollte eine eingegebene Zeichenkettenkonstante, die ein oder mehrere Kommata enthält, in doppelten Anführungszeichen gesetzt werden. Die doppelten Anführungszeichen stellen sicher, daß die Zeichenkette als Einheit behandelt und nicht in zwei oder mehrere Teile aufgeteilt wird.

Wenn der Benutzer eine der drei letzten Regeln verletzt, gibt BASIC die Fehlermeldung `Korrigieren Sie die Eingabe` aus. Die Meldung wird solange wiederholt, bis die Eingabe sowohl in Anzahl als auch im Typ mit der Variablenliste übereinstimmt.

Wenn Sie die Eingabeaufforderung informativer als ein Fragezeichen gestalten möchten, kann diese wie folgt erstellt werden:

```
PRINT "Wie lautet die korrekte Uhrzeit (Std., Min.)";
INPUT "(Std,Min)"; Std$, Min$
```

Darauf folgt die Aufforderung:

```
Wie lautet die korrekte Uhrzeit (Std., Min.)?
```

Beachten Sie das Semikolon zwischen der Eingabeaufforderung und den Eingabevariablen. Dieses Semikolon bewirkt das Erscheinen eines Fragezeichens als Teil der Eingabeaufforderung. Es kann vorkommen, daß Sie das Fragezeichen unterdrücken möchten; in diesem Fall ist ein Komma zwischen der Eingabeaufforderung und der Variablenliste einzusetzen:

```
INPUT "Geben Sie die Uhrzeit ein (Std., Min.): ", Std$, Min$
```

Dies ergibt folgende Aufforderung:

```
Geben Sie die Uhrzeit ein (Std., Min.):
```

3.2.2 Die LINE INPUT-Anweisung

Wenn das Programm Textzeilen mit eingeschlossenen Kommata, führenden oder nachfolgenden Leerzeichen akzeptieren soll und außerdem der Benutzer nicht ständig daran erinnert werden soll, die Eingabe in doppelte Anführungszeichen zu setzen, ist die Anweisung **LINE INPUT** zu verwenden. Die Anweisung **LINE INPUT** akzeptiert, wie der Name schon sagt, eine Eingabezeile (die durch Betätigung der EINGABETASTE abgeschlossen wird) von der Tastatur und speichert diese in einer einzigen Zeichenkettenvariablen. Zum Unterschied von **INPUT**, gibt die **LINE INPUT**-Anweisung standardmäßig bei einer Eingabeaufforderung kein Fragezeichen aus, erlaubt jedoch die Ausgabe einer Zeichenkette für die Eingabeaufforderung.

Nachstehendes Beispiel verdeutlicht den Unterschied zwischen **INPUT** und **LINE INPUT**:

```
' Weise die Eingabe drei unterschiedlichen Variablen zu:
PRINT "Geben Sie drei durch Kommata getrennte Werte ein: ",
INPUT "", A$, B$, C$

' Weise die Eingabe einer Variablen zu (Kommata werden nicht
' als Trenner zwischen Eingaben behandelt):
LINE INPUT "Geben Sie die gleichen drei Werte ein: ", D$

PRINT "A$ = "; A$
PRINT "B$ = "; B$
PRINT "C$ = "; C$
PRINT "D$ = "; D$
```

Ausgabe

```
Geben Sie drei durch Kommata getrennte Werte ein:
Esel, Katze, und Hund
Geben Sie die gleichen drei Werte ein: Esel, Katze, und Hund
A$ = Esel
B$ = Katze
C$ = und Hund
D$ = Esel, Katze und Hund
```

Sowohl mit **INPUT** als auch mit **LINE INPUT** wird eine Eingabe abgeschlossen, wenn der Benutzer die EINGABETASTE betätigt, was den Cursor zusätzlich auf die nächste Zeile bewegt. Wie das nächste Beispiel zeigt, hält ein Semikolon zwischen dem Schlüsselwort **INPUT** und der Eingabeaufforderung den Cursor auf derselben Zeile:

```
INPUT "Erster Wert: ", A
INPUT; "Zweiter Wert: ", B
INPUT "    Dritter Wert: ", C
```

Folgender Ausschnitt zeigt einige Mustereingaben für das vorherige Programm und die Positionen der Aufforderungen:

```
Erster Wert: 5
Zweiter Wert: 4      Dritter Wert: 3
```

3.2.3 Die INPUT$-Funktion

Sowohl **INPUT** als auch **LINE INPUT** veranlassen das Programm, auf die Betätigung der EINGABETASTE durch den Benutzer zu warten, bevor sie das Eingegebene speichern; das heißt, daß sie eine Eingabezeile lesen und diese anschließend den Programmvariablen zuweisen. Im Gegensatz dazu wartet die Funktion **INPUT$**(*Zahl*) nicht auf die Betätigung der EINGABETASTE; stattdessen liest sie eine gegebene Anzahl von Zeichen. Die folgende Zeile eines Programmes, zum Beispiel, liest drei vom Benutzer getippte Zeichen und speichert die Zeichenkette mit drei Zeichen in der Variablen `Test$`:

```
Test$ = INPUT$(3)
```

Im Gegensatz zur **INPUT**-Anweisung fordert die **INPUT$**-Funktion den Benutzer weder zur Eingabe von Daten auf, noch gibt sie eingegebene Zeichen auf den Bildschirm aus. Da **INPUT$** darüberhinaus eine Funktion darstellt, kann sie nicht selbständig als komplette Anweisung stehen. **INPUT$** muß wie folgt in einem Ausdruck erscheinen:

```
INPUT x                    ' INPUT ist eine Anweisung.
PRINT INPUT$(1)            ' INPUT$ ist eine Funktion, sie muß
                           ' daher in einem Ausdruck
Y$ = INPUT$(1)             ' erscheinen.
```

Die Funktion **INPUT$** liest Eingaben von der Tastatur als eine unformatierte Folge von Zeichen. Anders als **INPUT** oder **LINE INPUT**, akzeptiert **INPUT$** jegliche betätigte Taste, einschließlich der Steuerungstasten, wie RÜCKTASTE und ESC-Taste. Das fünfmalige Betätigen der EINGABETASTE weist, beispielsweise, der Variablen `Test$` fünf Wagenrücklaufzeilen in der nächsten Zeile zu:

```
Test$ = INPUT$(5)
```

3.2.4 Die INKEY$-Funktion

Die **INKEY$**-Funktion vervollständigt die Liste der BASIC-Funktionen und -Anweisungen für Tastatureingaben. Wenn BASIC einen Ausdruck antrifft, der die **INKEY$**-Funktion enthält, überprüft es, ob der Benutzer nach den folgenden Punkten eine Taste betätigt hat:

- seit dem letzten Finden eines Ausdruckes mit **INKEY$**
- seit Programmbeginn, beim ersten Auftreten von **INKEY$**

Wenn seit der letzten Überprüfung keine Taste betätigt wurde, gibt **INKEY$** eine leere Zeichenkette ("") aus. Wenn eine Taste betätigt wurde, gibt **INKEY$** das dieser Taste entsprechende Zeichen aus.

Beispiel

Der wichtigste Unterschied zwischen **INKEY$** und den anderen in diesem Abschnitt erläuterten Anweisungen und Funktionen besteht in der Tatsache, daß **INKEY$** während der Überprüfung der Eingabe das Programm mit der Ausführung anderer Vorgänge fortfahren läßt. Im Gegensatz dazu halten **INPUT**, **LINE INPUT** und **INPUT$** die Programmausführung bis zur nächsten Eingabe an, wie im nachstehenden Beispiel veranschaulicht:

```
PRINT "Beliebige Taste zum Starten und zum Beenden ";
PRINT "betätigen."
' Tue nichts, bis der Benutzer
' eine Taste betätigt hat:
Beginn$ = INPUT$(1)
I = 1
' Gib die Zahlen von eins bis eine Million aus.
' Überprüfe, ob eine Taste während der Schleifenausführung
' betätigt wird:
DO
   PRINT I
   I = I + 1
' Durchlaufe die Schleife, bis der Wert der Variablen I
' größer als eine Million ist, oder bis eine Taste
' betätigt wird:
LOOP UNTIL I > 1000000 OR INKEY$ <> ""
```

3.3 Steuerung des Text-Cursors

Bei der Ausgabe von geschriebenem Text auf den Bildschirm markiert der Text-Cursor die Stelle auf dem Bildschirm, an der eine Ausgabe des Programmes - oder vom Benutzer eingetippte Eingaben - als nächstes erscheinen werden. Im nächsten Beispiel wartet der Cursor auf Zeile 1 in Spalte 10 auf eine Eingabe, nachdem die Anweisung **INPUT** ihre aus neun Zeichen bestehende Eingabeaufforderung, `Vorname:`, ausgegeben hat:

```
' Lösche den Bildschirm und beginne Ausgabe in Zeile 1,
' Spalte 1:
CLS
INPUT "Vorname: ", VorName$
```

Im nächsten Beispiel hält das Semikolon am Ende der zweiten **PRINT**-Anweisung den Cursor auf Zeile 2 in Spalte 32 fest:

```
CLS
PRINT
' Einunddreißig Zeichen sind in der nächsten Zeile:
PRINT "Betätigen Sie beliebige Taste zum Fortfahren.";
PRINT INPUT$(1)
```

Abschnitte 3.3.1 bis 3.3.3 zeigen, wie die Position des Cursors gesteuert und seine Form verändert wird, und wie Sie Informationen über seine Position erhalten.

3.3.1 Positionieren des Cursors

Die bisher erläuterten Anweisungen und Funktionen für Ein- und Ausgabe erlauben keine gute Kontrolle über die Anzeige einer Ausgabe oder über die Positionierung des Cursors nach Erscheinen der Ausgabe. Eingabeanfragen oder Ausgaben beginnen immer in der äußersten linken Spalte des Bildschirms und erstrecken sich jeweils über eine Zeile von oben nach unten, solange nicht ein Semikolon in den **PRINT**- oder **INPUT**-Anweisungen eingesetzt wird, um die Sequenz Wagenrücklauf-Zeilenvorschub zu unterdrücken.

Die in Abschnitt 3.1.4, "Überspringen von Leerzeichen und Springen in eine angegebene Spalte", beschriebenen Anweisungen **SPC** und **TAB** ermöglichen eine bessere Kontrolle über die Position des Cursors, indem sie die Bewegung des Cursors in jeder Spalte innerhalb einer gegebenen Zeile erlauben.

Die Anweisung **LOCATE**, die folgende Syntax aufweist, erweitert diese Kontrolle um einen weiteren Schritt:

LOCATE [*Zeile*][,[*Spalte*][,[*Cursor*][,[*Beginn*][, *Ende*]]]]

Die **LOCATE**-Anweisung ermöglicht die Positionierung des Cursors in jeder *Zeile* oder *Spalte*, wie aus der Ausgabe des nächsten Beispiels ersichtlich:

```
CLS
FOR Zeile = 9 TO 1 STEP -2
   Spalte = 2 * Zeile
   LOCATE Zeile, Spalte
   PRINT "12345678901234567890";
NEXT
```

Ausgabe

```
12345678901234567890
    12345678901234567890
        12345678901234567890
            12345678901234567890
                12345678901234567890
```

3.3.2 Ändern der Cursor-Form

Die in der **LOCATE**-Syntax gezeigten optionalen Argumente *Cursor*, *Beginn* und *Ende* erlauben auch die Änderung der Cursor-Form und machen den Cursor sichtbar oder unsichtbar. Ein Wert von 1 für *Cursor* macht diesen sichtbar, während ihn ein Wert von 0 unsichtbar macht. Die Argumente *Beginn* und *Ende* steuern die Höhe des Cursors, wenn dieser eingeschaltet ist, indem sie die oberen und unteren Bildpunktzeilen für den Cursor angeben. (Jedes Zeichen auf dem Bildschirm wird aus Zeilen von Bildpunkten (Pixel) zusammengesetzt, die Lichtpunkte auf dem Bildschirm darstellen.) Wenn sich ein Cursor über die Höhe einer Textzeile erstreckt, hat die Bildpunktzeile des oberen Cursorrandes den Wert 0, während die Bildpunktzeile des unteren Cursorrandes einen Wert von 7 oder 13, je nach dem Typ des Bildschirms/Adapters (monochrom ist 13, farbig ist 7) aufweist.

Der Cursor kann eingeschaltet und seine Form verändert werden, ohne eine neue Position für ihn anzugeben. Die nächste Anweisung, zum Beispiel, hält den Cursor dort fest, wo er sich nach Abschluß der nächsten **PRINT**- oder **INPUT**-Anweisung befindet und verändert ihn auf die Höhe eines halben Zeichens:

```
LOCATE , , 1, 2, 5  ' Die Argumente für Zeile und Spalte
                    ' sind beide optional.
```

Folgende Beispiele zeigen unterschiedliche Cursor-Formen, die anhand unterschiedlicher *Beginn*- und *Ende*-Werte auf einem Farbmonitor erzeugt werden. Jeder in der linken Spalte gezeigten **LOCATE**-Anweisung folgt die Anweisung

```
INPUT "ANFRAGE:", X$
```

Anweisung	***Eingabeaufforderung und Cursor-Form***
`LOCATE ,, 1, 0, 7`	EINGABEAUFFORDERUNG: ▌
`LOCATE ,, 1, 4, 7`	EINGABEAUFFORDERUNG: ▖
`LOCATE ,, 1, 4, 7`	EINGABEAUFFORDERUNG: ▖
`LOCATE ,, 1, 6, 2`	EINGABEAUFFORDERUNG: ▚

Wie Sie in den vorhergehenden Beispielen feststellen können, hat die Angabe eines *Beginn*-Arguments, das größer ist als das *Ende*-Argument, einen zweiteiligen Cursor zur Folge.

3.3.3 Informationen über die Position des Cursors

Die Funktionen **CSRLIN** und **POS**(*n*) können als das Gegenstück der Anweisung **LOCATE** bezeichnet werden; während **LOCATE** dem Cursor angibt, wohin er springen soll, informieren **CSRLIN** und **POS**(*n*) das Programm über die Position des Cursors. Im einzelnen gibt **CSRLIN** die aktuelle Zeile und **POS**(*n*) die aktuelle Spalte der Cursor-Position an.

Das Argument *n* von **POS**(*n*) wird als "Ersatz"-Argument bezeichnet; das heißt, *n* ist ein Platzhalter, der ein beliebiger numerischer Ausdruck sein kann. Zum Beispiel geben sowohl `POS(0)` als auch `POS(1)` den gleichen Wert an.

Beispiel

Das folgende Beispiel verwendet die Funktion **POS**(*n*), um 50 Sterne in Zeilen zu je 13 Sternen auszugeben.

```
FOR I% = 1 TO 50
   PRINT "*";                  ' Gib einen Stern aus und halte
                               ' den Cursor auf derselben
                               ' Zeile fest.
   IF POS(1) > 13 THEN PRINT   ' Wenn sich der Cursor nach
                               ' Spalte 13 befindet,
                               ' gehe zur nächsten Zeile.
NEXT
```

Ausgabe

```
*************
*************
*************
***********
```

3.4 Arbeiten mit den Datendateien

Datendateien sind physikalisch vorhandene Bereiche auf Ihrer Diskette/Festplatte in denen Informationen permanent gespeichert sind. Folgende drei Aufgaben werden durch die Verwendung von Datendateien in BASIC-Programmen beträchtlich vereinfacht:

1. Erstellen, Manipulieren und Speichern großer Mengen von Daten.
2. Zugriff auf verschiedene Datensätze anhand eines einzigen Programms.
3. Benutzung desselben Datensatzes in mehreren unterschiedlichen Programmen.

Die nächsten Abschnitte stellen die Konzepte von Datensätzen und -feldern vor, sowie unterschiedliche Möglichkeiten, auf Datendateien mit Hilfe von BASIC zuzugreifen. Nach Abschluß der Abschnitte 3.4.1 bis 3.4.7, werden Sie in der Lage sein, folgende Aufgaben zu bewältigen:

- Erstellen neuer Datendateien.
- Öffnen existierender Dateien und Lesen ihrer Inhalte.
- Ergänzen einer existierenden Datendatei mit neuen Informationen.
- Verändern der Inhalte einer existierenden Datendatei.

3.4.1 Organisation der Datendateien

Eine Datendatei ist eine Ansammlung von zusammengehörenden Informationsblöcken, auch "Datensätze" genannt. Jeder Datensatz (Record) in einer Datendatei ist seinerseits in "Datenfelder" oder in regelmäßig wiederkehrende Informationsgrößen innerhalb jedes Datensatzes unterteilt. Wird eine Datendatei mit einer altmodischeren Möglichkeit der Informationsspeicherung verglichen - zum Beispiel einem Aktenordner, der die in einer Firma von Stellenbewerbern ausgefüllten Personalfragebögen enthält - dann entspricht ein Datensatz einem Fragebogen in diesem Aktenordner. Um den Vergleich einen Schritt weiterzuführen, entspricht ein Feld einer Informationsgröße auf jedem Fragebogen, wie zum Beispiel der Sozialversicherungsnummer.

Hinweis Wenn Sie nicht unter Verwendung von Datensätzen auf eine Datei zugreifen möchten, sondern diese stattdessen als eine unformatierte Folge von Bytes behandeln, lesen Sie bitte Abschnitt 3.4.7, "Binärdatei-E/A".

3.4.2 Sequentielle und Direktzugriffsdateien

Die Ausdrücke "sequentielle Datei" und "Direktzugriffsdatei" (Random Access) beziehen sich auf zwei unterschiedliche Möglichkeiten, anhand von BASIC-Programmen Daten auf Diskette/Festplatte zu speichern bzw. auf Daten von Diskette/Festplatte zuzugreifen. Eine einfache Möglichkeit, sich diese beiden Arten von Dateien vorzustellen, besteht im folgenden Vergleich: eine sequentielle Datei ist wie ein Tonband, während eine Direktzugriffsdatei einer Langspielplatte ähnelt. Um ein Lied auf einem Tonband zu finden, muß das Band vom Anfang an sequentiell bis zum gesuchten Lied vorgespult werden, es gibt keine Möglichkeit, das gewünschte Lied direkt anzuspringen. Dies gleicht der Art und Weise, in der Informationen in einer sequentiellen Datei zu finden sind: um den 500-sten Datensatz zu finden, müssen die Datensätze 1 bis 499 gelesen werden.

Wenn Sie dagegen ein besonderes Lied auf einer LP hören möchten, genügt es, den Tonarm des Plattenspielers anzuheben und die Nadel direkt vor dem Lied aufzusetzen: Sie haben dadurch direkten Zugriff auf der Platte, ohne zunächst alle Lieder vor dem gewünschten Lied anhören zu müssen. Auf der gleichen Weise kann jeder gewünschte Datensatz aus einer Direktzugriffsdatei durch Angabe seiner Nummer aufgerufen werden. Die Zugriffszeit läßt sich durch dieses Verfahren erheblich reduzieren.

Hinweis Obwohl es keine Möglichkeit gibt, direkt zu einem bestimmten Datensatz in einer sequentiellen Datei zu springen, können Sie mit Hilfe der Anweisung **SEEK** direkt auf ein bestimmtes Byte in der Datei springen (dies entspricht dem "Vorspulen" und "Zurückspulen" des Bandes, um die vorhergehende Analogie weiterzuführen). Nähere Informationen über diesen Vorgang finden Sie in Abschnitt 3.4.7, "Binärdatei-E/A".

3.4.3 Öffnen einer Datendatei

Bevor das Programm in der Lage ist, eine Datendatei zu lesen, zu verändern oder mit Daten zu ergänzen, muß es zunächst die Datei öffnen. BASIC verwendet dafür die Anweisung **OPEN**, die auch das Anlegen einer neuen Datei ermöglicht. Nachstehende Liste beschreibt die unterschiedlichen Anwendungen der Anweisung **OPEN**:

- Erstellen und Öffnen einer neuen Datendatei, so daß ihr Datensätze hinzugefügt werden können:

```
' Es befindet sich im aktuellen Verzeichnis keine Datei
' mit dem Namen preis.dat :
OPEN "preis.dat" FOR OUTPUT AS #1
```

- Öffnen einer existierenden Datendatei, so daß neue Datensätze bereits in der Datei vorhandene Daten überschreiben:

```
' Eine Datei mit dem Namen preis.dat befindet sich
' bereits in dem aktuellen Verzeichnis; es können neue
' Datensätze in sie geschrieben werden, wobei jedoch alle
' alten Datensätze verloren gehen:
OPEN "preis.dat" FOR OUTPUT AS #1
```

- Öffnen einer existierenden Datendatei, so daß neue Datensätze an das Ende der Datei hinzugefügt werden können, damit die bereits in der Datei befindlichen Daten erhalten bleiben:

  ```
  OPEN "preis.dat" FOR APPEND AS #1
  ```

  Der **APPEND**-Modus erstellt eine neue Datei mit dem angegebenen Namen, wenn diese in dem aktuellen Verzeichnis noch nicht existiert.

- Öffnen einer existierenden Datendatei, so daß alte Datensätze daraus gelesen werden können:

  ```
  OPEN "preis.dat" FOR INPUT AS #1
  ```

  Weitere Informationen zu den Modi **INPUT**, **OUTPUT** und **APPEND** sind Abschnitt 3.4.5, "Die Verwendung von sequentiellen Dateien" zu entnehmen.

- Öffnen einer existierenden Datendatei (oder Erstellen einer neuen, wenn eine Datei mit diesem Namen nicht existiert), anschließend Lesen oder Schreiben von Datensätzen fester Länge in oder aus der Datei:

  ```
  OPEN "preis.dat" FOR RANDOM AS #1
  ```

  Weitere Informationen über diesen Modus finden Sie in Abschnitt 3.4.6, "Die Verwendung von Direktzugriffsdateien".

- Öffnen einer existierenden Datendatei (oder Erstellen einer neuen, wenn eine Datei mit diesem Namen nicht existiert), anschließend Lesen von Daten aus der Datei oder Ergänzen der Datei mit neuen Daten an einer beliebigen Byte-Position in der Datei:

  ```
  OPEN "preis.dat" FOR BINARY AS #1
  ```

  Weitere Informationen über diesen Modus entnehmen Sie bitte Abschnitt 3.4.7, "Binärdatei-E/A".

3.4.3.1 Dateinummern in BASIC

Die Anweisung **OPEN** umfaßt mehr als die Angabe eines Daten-E/A-Modus für eine bestimmte Datei (**OUTPUT**, **INPUT**, **APPEND**, **RANDOM** oder **BINARY**); sie verknüpft ebenfalls eine einmalige Dateinummer mit dieser Datei. Anschließend wird diese, aus jeder Ganzzahl von 1 bis 255 bestehende Dateinummer, von nachfolgenden Datei-E/A-Anweisungen im Programm als Kurzschreibweise zur Bezugnahme auf die Datei verwendet. Solange die Datei geöffnet ist, bleibt die Nummer mit der Datei verknüpft. Beim Schließen der Datei wird die Dateinummer zur weiteren Verwendung mit anderen Dateien freigegeben. (In Abschnitt 3.4.4 finden Sie nähere Informationen über das Schließen von Dateien.) Die BASIC-Programme können mehr als eine Datei gleichzeitig öffnen.

Mit Hilfe der Funktion **FREEFILE** läßt sich eine unbenutzte Dateinummer finden.

FREEFILE gibt die nächste verfügbare Nummer an, die in einer **OPEN**-Anweisung mit einer Datei verknüpft werden kann. **FREEFILE** würde zum Beispiel den Wert 3 nach folgenden **OPEN**-Anweisungen angeben:

```
OPEN "test1.dat" FOR RANDOM AS #1
OPEN "test2.dat" FOR RANDOM AS #2
DateiNum = FREEFILE
OPEN "test3.dat" FOR RANDOM AS #DateiNum
```

Die Funktion **FREEFILE** ist besonders beim Erstellen einer eigenen Bibliothek hilfreich, die Prozeduren zum Öffnen von Dateien enthält. Mit **FREEFILE** ist es nicht erforderlich, Informationen über die Anzahl offener Dateien an die Prozeduren zu übergeben.

3.4.3.2 Dateinamen in BASIC

Jeder Zeichenkettenausdruck, der aus jeglicher Kombination nachstehender Zeichen besteht, kann als Dateiname in **OPEN**-Anweisungen verwendet werden:

- Die Buchstaben a-z und A-Z
- Die Zahlen 0-9
- Die folgenden Sonderzeichen:
  ' ( ) { } @ # $ % ^ & ! - _ ' ~

Der Zeichenkettenausdruck kann sowohl eine optionale Laufwerksangabe als auch eine vollständige oder teilweise Pfadangabe enthalten. Daher kann das BASIC-Programm mit Datendateien in unterschiedlichen Laufwerken oder Verzeichnissen arbeiten. Alle folgenden **OPEN**-Anweisungen sind zum Beispiel gültig:

```
OPEN "..\rang.qtr" FOR INPUT AS #1

OPEN "a:\gehälter\1987.man" FOR INPUT AS #2

DateiName$ = "TempDat"
OPEN DateiName$ FOR OUTPUT AS #3

BasisName$ = "Inventur"
OPEN BasisName$ + ".dat" FOR OUTPUT AS #4
```

DOS legt eigene Einschränkungen für Dateinamen auf: Es können nicht mehr als acht Zeichen für den Basisnamen (links vom optionalen Punkt) und nicht mehr als drei Zeichen für die Erweiterung (rechts vom optionalen Punkt) verwendet werden. Lange Dateinamen werden in BASIC wie folgt gekürzt:

Dateiname im Programm	***Daraus folgender Dateiname in DOS***
`PROG@DAT@DATEI`	`PROG@DAT.@DA` Der BASIC-Name ist länger als 11 Zeichen. BASIC nimmt daher die ersten 8 Buchstaben für den Basisnamen, fügt einen Punkt (.) ein und verwendet die nächsten drei Buchstaben als Erweiterung. Alle weiteren Zeichen werden abgeschnitten.
`Post#.Version1`	`post#.ver` Der Basisname (`Post#`) ist kürzer als acht Zeichen, aber die Erweiterung (`Version1`) ist länger als 3 Zeichen, daher wird die Erweiterung auf drei Zeichen gekürzt.
`post_daten.bak`	Führt zu der Laufzeitfehlermeldung `Unzulässiger Dateiname`. Der Basisname muß kürzer als acht Zeichen sein, wenn eine ausdrückliche Erweiterung verwendet werden soll (in diesem Fall `.bak`).

DOS arbeitet unabhängig von Groß-/Kleinschreibung, so daß alle Kleinbuchstaben in Dateinamen in Großbuchstaben umgewandelt werden. Deshalb sollten Dateien nicht aufgrund der gemischten Schreibweise von Klein- und Großbuchstaben unterschieden werden. Gibt es beispielsweise auf der Diskette bereits eine Datei *mess.dat*, würde folgende **OPEN**-Anweisung diese Datei überschreiben und jegliche darin gespeicherte Information zerstören:

```
OPEN "Mess.Dat" FOR OUTPUT AS #1
```

3.4.4 Schließen von Datendateien

Das Schließen einer Datendatei hat zwei wichtige Auswirkungen: erstens schreibt es alle aktuell im Dateipuffer (temporärer Bereich im Speicher) befindlichen Daten in die Datei; zweitens gibt es die mit der Datei verknüpfte Dateinummer zur Benutzung mit einer anderen **OPEN**-Anweisung frei.

Eine Datei ist innerhalb eines Programmes anhand der Anweisung **CLOSE** zu schließen. Wenn zum Beispiel die Datei *preis.dat* mit der folgenden Anweisung geöffnet ist:

```
OPEN "preis.dat" FOR OUTPUT AS #1
```

dann beendet die Anweisung `CLOSE #1` die Ausgabe auf *preis.dat*. Wenn *preis.dat* mit

```
OPEN "preis.dat" FOR OUTPUT AS #2
```

geöffnet wird, eignet sich die Anweisung `CLOSE #2` zur Beendigung der Ausgabe. Eine **CLOSE**-Anweisung ohne Dateinummernargumente schließt alle offenen Dateien.

Eine Datendatei wird ebenfalls unter folgenden Bedingungen geschlossen:

- Das BASIC-Programm, das E/A durchführt, wird beendet (die Programmbeendigung schließt immer alle offenen Datendateien).
- Das Programm, das E/A durchführt, überträgt mit der Anweisung **RUN** die Kontrolle an ein anderes Programm.

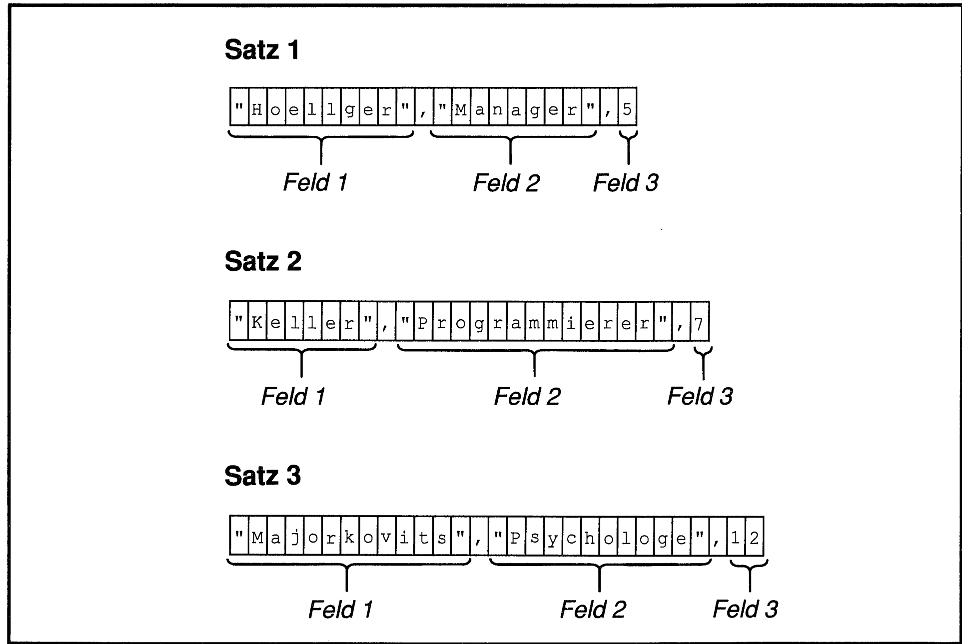
3.4.5 Die Verwendung von sequentiellen Dateien

Dieser Abschnitt beschreibt die Organisation von Datensätzen in sequentiellen Datendateien und zeigt anschließend, wie Daten aus einer offenen sequentiellen Datei gelesen bzw. in eine offene sequentielle Datei geschrieben werden.

3.4.5.1 Datensätze in sequentiellen Dateien

Sequentielle Dateien sind ASCII(Text)-Dateien. Das heißt, daß eine sequentielle Datei anhand jedes Textverarbeitungsprogrammes betrachtet oder verändert werden kann. Datensätze werden in sequentiellen Dateien als einzelne, mit einer Wagenrücklauf-Zeilenvorschub (WR-ZV)-Sequenz beendete Textzeile gespeichert. Jeder Datensatz ist in Felder eingeteilt, oder in sich wiederholenden Datenstücken, die in jedem Datensatz in der gleichen Reihenfolge erscheinen. Abbildung 3.2 veranschaulicht, wie drei Datensätze in einer sequentiellen Datei erscheinen könnten.

Abbildung 3.2 Datensätze in sequentiellen Dateien



Es ist zu beachten, daß jeder Datensatz einer sequentiellen Datei eine andere Länge aufweisen kann und daß darüber hinaus Felder in verschiedenen Datensätzen unterschiedliche Längen haben können.

Die Art der Variablen, in der ein Feld gespeichert ist, bestimmt den Anfang und das Ende dieses Feldes. (Beispiele für das Schreiben in Datensätze bzw. das Lesen aus Datensätzen finden Sie in Abschnitt 3.4.5.2 bis 3.4.5.6.) Wenn das Programm zum Beispiel ein Feld in eine Zeichenkettenvariable einliest, kann das Ende dieses Feldes wie folgt angezeigt werden:

- Doppelte Anführungszeichen ("), wenn die Zeichenkette mit doppelten Anführungszeichen beginnt.
- Komma (,), wenn die Zeichenkette nicht mit doppelten Anführungszeichen beginnt.
- WR-ZV, wenn sich das Feld am Ende des Datensatzes befindet.

Wenn das Programm dagegen ein Feld in eine numerische Variable einliest, kann das Ende dieses Feldes wie folgt angezeigt werden:

- Komma
- Ein oder mehrere Leerzeichen
- WR-ZV

3.4.5.2 Ergänzen einer sequentiellen Datei mit Daten

Einer neuen sequentiellen Datei können Daten hinzugefügt werden, nachdem diese zunächst anhand einer **OPEN** *Dateiname* **FOR OUTPUT**-Anweisung zur Aufnahme von Datensätzen geöffnet wird. Mit Hilfe der **WRITE #**-Anweisung lassen sich Datensätze in eine Datei schreiben.

Sequentielle Dateien können entweder zum Schreiben oder zum Lesen geöffnet werden, aber nicht beides gleichzeitig. Wenn Sie in eine sequentielle Datei geschrieben haben und die gespeicherten Daten wieder einlesen möchten, müssen Sie die Datei zunächst schließen und danach wieder für Eingabe öffnen.

Beispiel

Das folgende kurze Programm erstellt eine sequentielle Datei mit dem Namen *preis.dat* und fügt dieser anschließend per Tastatur eingegebene Daten hinzu. Die **OPEN**-Anweisung in diesem Programm erstellt die Datei und bereitet sie auch zur Aufnahme von Datensätzen vor. Danach schreibt **WRITE #** jeden Datensatz in die Datei. Beachten Sie, daß die in der **WRITE #**-Anweisung verwendete Nummer mit der dieser Datei *preis.dat* in der **OPEN**-Anweisung zugewiesenen Nummer übereinstimmt.

```
' Erstelle eine Datei mit dem Namen preis.dat und öffne sie
' zur Aufnahme von neuen Daten:
OPEN "preis.dat" FOR OUTPUT AS #1
DO
   ' Fahre fort, neue Datensätze in preis.dat zu schreiben,
   ' bis der Benutzer die Eingabetaste betätigt, ohne einen
   ' Firmennamen einzugeben:
   PRINT "Firma (Betätigen Sie die <Eingabetaste> zum";
   INPUT "Beenden): ", Firma$
   IF Firma$ <> "" THEN
      ' Eingabe der anderen Felder des Datensatzes:
      INPUT "Art: ",      Art$
      INPUT "Größe: ",    Gross$
      INPUT "Farbe: ",    Farbe$
      INPUT "Menge: ",    Menge
      ' Schreibe den neuen Datensatz mit der Anweisung WRITE
      ' # in die Datei:
      WRITE #1, Firma$, Art$, Gross$, Farbe$, Menge
   END IF
```

```
LOOP UNTIL Firma$ = ""
' Schließe preis.dat (Ausgabe in die Datei wird damit
' beendet):
CLOSE #1
END
```

Warnung Wenn Sie im vorhergehenden Beispiel bereits eine Datei mit dem Namen *preis.dat* auf der Diskette hatten, würde der in der **OPEN**-Anweisung angegebene Modus **OUTPUT** den bereits vorhandenen Inhalt von *preis.dat* vor Eingabe jeglicher neuer Daten löschen. Wenn Sie neue Daten an das Ende einer bereits vorhandenen Datei anfügen möchten, ohne deren alten Inhalt zu löschen, ist der **APPEND**-Modus von **OPEN** zu verwenden. Weitere Informationen über diesen Modus finden Sie in Abschnitt 3.4.5.4, "Ergänzen einer existierenden sequentiellen Datei mit Daten".

3.4.5.3 Lesen von Daten aus einer sequentiellen Datei

Aus einer sequentiellen Datei können Daten nach Öffnen der Datei mit Hilfe der Anweisung **OPEN** *Dateiname* **FOR INPUT** gelesen werden. Verwenden Sie die Anweisung **INPUT #**, um Datensätze aus der Datei Feld für Feld zu lesen. (Weitere Informationen zu anderen, mit einer sequentiellen Datei zu verwendenden Dateieingabe-Anweisungen und -Funktionen finden Sie in Abschnitt 3.4.5.6, "Weitere Möglichkeiten, Daten aus einer sequentiellen Datei zu lesen".)

Beispiel

Das folgende Programm öffnet die im vorhergehenden Beispiel erstellte Datendatei *preis.dat* und liest Datensätze aus der Datei, wobei der gesamte Datensatz auf dem Bildschirm angezeigt wird, wenn die Menge für den Artikel kleiner als der eingegebene Betrag ist.

Die Anweisung `INPUT #1` liest einen Datensatz Feld für Feld aus *preis.dat*, indem sie die Felder des Datensatzes den Variablen `Firma$, Art$, Gross$, Farbe$` und `Menge` zuweist. Da es sich um eine sequentielle Datei handelt, werden die Datensätze nacheinander vom ersten bis zum letzten eingetragenen Datensatz gelesen.

Die Funktion **EOF** (Dateiende) überprüft, ob der letzte Datensatz von **INPUT #** gelesen wurde. Wenn das der Fall ist, gibt **EOF** den Wert -1 (wahr) zurück und die Schleife zum Lesen der Daten wird beendet; wenn der letzte Datensatz nicht gelesen wurde, gibt **EOF** den Wert 0 (falsch) zurück und es wird der nächste Datensatz aus der Datei gelesen.

```
OPEN "preis.dat" FOR INPUT AS #1

PRINT "Anzeige aller Artikel unterhalb welchen Niveaus";
INPUT ""; NeuBest

DO UNTIL EOF(1)
   INPUT #1, Firma$, Art$, Gross$, Farbe$, Menge
   IF Menge < NeuBest THEN
      PRINT  Firma$, Art$, Gross$, Farbe$, Menge
   END IF
LOOP

CLOSE #1
END
```

3.4.5.4 Ergänzen einer sequentiellen Datei mit Daten

Wie bereits früher erwähnt, kann eine sequentielle Datei nicht einfach im Ausgabemodus geöffnet werden, wenn Sie eine sequentielle Datei auf Diskette haben und weitere Daten an deren Ende anfügen möchten, da deren aktuelle Inhalte dadurch zerstört werden. Stattdessen ist der Anfügemodus **APPEND** wie im nächsten Beispiel zu verwenden.

```
OPEN "preis.dat" FOR APPEND AS #1
```

Tatsächlich ist **APPEND** immer eine sichere Alternative zu **OUTPUT**, da der Anfügemodus eine neue Datei erstellt, wenn keine Datei mit dem angegebenen Namen vorhanden ist. Befindet sich, zum Beispiel, eine Datei mit dem Namen *preis.dat* nicht auf der Diskette, wird eine neue Datei mit diesem Namen durch oben aufgeführte Musteranweisung angelegt.

3.4.5.5 Weitere Möglichkeiten, Daten in eine sequentielle Datei zu schreiben

Die vorhergehenden Beispiele benutzten alle die Anweisungen **WRITE #** zum Schreiben von Datensätzen in eine sequentielle Datei. Es gibt jedoch eine weitere Anweisung, **PRINT #**, zum Schreiben von Datensätzen in sequentielle Dateien.

Die beste Möglichkeit, den Unterschied zwischen diesen beiden Anweisungen zur Datenspeicherung zu unterstreichen, besteht darin, den Inhalt einer mit beiden Anweisungen erstellten Datei zu untersuchen. Der folgende kurze Ausschnitt öffnet eine Datei *test.dat* und schreibt den gleichen Datensatz zweimal, einmal mit **WRITE #** und einmal mit **PRINT #**, in sie. Nach Ablauf dieses Programms, können die Inhalte von *test.dat* mit dem DOS-Befehl **TYPE** untersucht werden.

```
OPEN "test.dat" FOR OUTPUT AS #1
Nm$ = "Penn, Willi"
Abt$ = "Benutzerschulung"
Niveau = 4
Alter = 25
WRITE #1, Nm$, Abt$, Niveau, Alter
PRINT #1, Nm$, Abt$, Niveau, Alter
CLOSE #1
```

Ausgabe in die Datei

```
"Penn, Willi","Benutzerschulung",4,25
Penn, Willi   Benutzerschulung            4              25
```

Der mit **WRITE #** gespeicherte Datensatz hat Kommata, die ausdrücklich jedes Feld des Datensatzes trennen, sowie Anführungszeichen, die jeden Zeichenkettenausdruck einschließen. Andererseits hat **PRINT #** ein Abbild des Datensatzes in die Datei geschrieben, das genauso aussieht, wie es mit einer einfachen **PRINT**-Anweisung auf dem Bildschirm erscheinen würde. Die Kommata in der **PRINT #**-Anweisung werden als "gehe zum nächsten Ausgabebereich" (ein neuer Ausgabebereich erscheint alle 14 Leerzeichen am Anfang einer Zeile) interpretiert, und die Zeichenkettenausdrücke sind nicht in Anführungszeichen eingeschlossen.

Es stellt sich jetzt die Frage, worin der Unterschied zwischen diesen Ausgabeanweisungen besteht, abgesehen vom Erscheinungsbild der Daten innerhalb der Datei. Der springenden Punkt ist der Vorgang, durch den das Programm Daten aus der Datei anhand einer **INPUT #**-Anweisung zurückliest. Im folgenden Beispiel liest das Programm den mit **WRITE #** gespeicherten Datensatz und gibt die Werte seiner Felder problemlos zurück:

```
OPEN "test.dat" FOR INPUT AS #1
' Lies den ersten Datensatz ein und zeige die Inhalte jedes
' Feldes an:
INPUT #1, Nm$, Abt$, Niveau, Alter
PRINT Nm$, Abt$, Niveau, Alter
```

```
' Lies den zweiten Datensatz ein und zeige die Inhalte jedes
' Feldes an:
INPUT #1, Nm$, Abt$, Niveau, Alter
PRINT Nm$, Abt$, Niveau, Alter
CLOSE #1
```

Ausgabe

```
Penn, Willi   Benutzerschulung           4       25
```

Wenn das Programm jedoch versucht, den nächsten mit **PRINT #** gespeicherten Datensatz einzulesen, führt dies zu der Fehlermeldung `Eingabe nach Dateiende`. Da das erste Feld nicht in doppelten Anführungszeichen steht, betrachtet die **INPUT #**-Anweisung das Komma zwischen `Penn` und `Willi` als Feldbegrenzer und weist daher der Variablen `Nm$` nur den Nachnamen `Penn` zu. Den Rest der Zeile liest **INPUT #** dann in die Variable `Abt$`. Da nun der gesamte Datensatz gelesen ist, bleibt nichts für das Lesen in die Variablen `Niveau` und `Alter` übrig. Das Ergebnis ist die Fehlermeldung `Eingabe nach Dateiende`.

Wenn Sie Datensätze mit Zeichenkettenausdrücken speichern, die Sie später anhand der Anweisung **INPUT #** lesen möchten, ist eine der folgenden Faustregeln zu beachten:

1. Speichern Sie die Datensätze anhand der Anweisung **WRITE #**.
2. Vergessen Sie nicht bei der Verwendung der Anweisung **PRINT #**, daß diese weder Kommata zum Trennen der Felder in den Datensatz schreibt noch Zeichenketten in Anführungszeichen einschließt. Sie müssen diese Feldtrenner selbst in die **PRINT #**-Anweisung schreiben.

Die Probleme im vorhergehenden Beispiel können beispielsweise durch die Verwendung von **PRINT #** mit Anführungszeichen, die jedes Feld umschließen, das Zeichenkettenausdrücke enthält, vermieden werden:

Beispiel

```
' 34 ist der ASCII-Wert für das doppelte Anführungszeichen:
Q$ = CHR$(34)
' Die nächsten vier Anweisungen schreiben alle den Datensatz
' in die Datei mit doppelten Anführungszeichen um jedes
' Zeichenkettenfeld:
PRINT #1, Q$ Nm$ Q$ Q$ Abt$ Q$ Niveau Alter
PRINT #1, Q$ Nm$ Q$;Q$ Abt$ Q$;Niveau;Alter
PRINT #1, Q$ Nm$ Q$,Q$ Abt$ Q$,Niveau,Alter
WRITE #1, Nm$, Abt$, Niveau, Alter
```

Ausgabe in die Datei

```
"Penn, Willi""Benutzerschulung" 4  25
"Penn, Willi""Benutzerschulung" 4  25
"Penn, Willi" "Benutzerschulung"         4            25
"Penn, Willi","Benutzerschulung",4,25
```

3.4.5.6 Weitere Möglichkeiten, Daten aus einer sequentiellen Datei zu lesen

In den vorhergehenden Abschnitten wird **INPUT #** dazu verwendet, einen Datensatz (eine Zeile von Daten einer Datei) zu lesen, um den nach **INPUT #** aufgeführten Variablen unterschiedliche Felder des Datensatzes zuzuweisen. Dieser Abschnitt untersucht alternative Möglichkeiten, Daten aus sequentiellen Dateien sowohl als Datensätze (**LINE INPUT #**) als auch als unformatierte Folgen von Bytes (**INPUT$**) zu lesen.

Die Anweisung LINE INPUT # Anhand der Anweisung **LINE INPUT #** kann das Programm eine Textzeile genauso lesen, wie sie in einer Datei erscheint, ohne Kommata oder Anführungszeichen als Feldbegrenzer zu interpretieren. Dies ist besonders hilfreich in Programmen, die mit ASCII-Textdateien arbeiten.

LINE INPUT # liest eine gesamte Zeile aus einer sequentiellen Datei (bis zu einer Sequenz Wagenrücklauf-Zeilenvorschub) in eine einzelne Zeichenkettenvariable.

Beispiele

Das folgende kurze Programm liest jede Zeile aus der Datei *kap1.txt* und schreibt diese Zeile anschließend auf den Bildschirm:

```
' Öffne kap1.txt für sequentielle Eingabe:
OPEN "kap1.txt" FOR INPUT AS #1

' Lies Zeilen solange sequentiell, bis keine mehr in der
' Datei vorhanden sind:
DO UNTIL EOF(1)

   ' Lies eine Zeile aus der Datei und speichere sie in der
   ' Variablen ZeilPuffer$:
   LINE INPUT #1, ZeilPuffer$

   ' Schreibe die Zeile auf den Bildschirm:
   PRINT ZeilPuffer$
LOOP
```

Das vorhergehende Programm kann leicht zu einem dateikopierenden Hilfsprogramm verändert werden, das jede aus der angegebenen Eingabedatei gelesene Zeile in eine andere Datei anstatt des Bildschirms schreibt:

```
' Geben Sie Namen von Eingabe- und Ausgabedateien an:
INPUT "zu kopierende Datei: ", DateiName1$
IF DateiName1$ = "" THEN END
INPUT "Name der neuen Datei: ", DateiName2$
IF DateiName2$ = "" THEN END
' Öffne erste Datei für sequentielle Eingabe:
OPEN DateiName1$ FOR INPUT AS #1
' Öffne zweite Datei für sequentielle Ausgabe:
OPEN DateiName2$ FOR OUTPUT AS #2
' Lies sequentiell Zeilen aus der ersten Datei, bis keine
' mehr in der Datei sind:
DO UNTIL EOF(1)
   ' Lies eine Zeile aus der ersten Datei und speichere sie
   ' in der Variablen ZeilPuffer$:
   LINE INPUT #1, ZeilPuffer$
   ' Schreibe ZeilPuffer$ in die zweite Datei:
   PRINT #2, ZeilPuffer$
LOOP
```

Die Funktion INPUT$** Eine weitere Möglichkeit Daten aus sequentiellen Dateien (letztlich aus jeder Datei) zu lesen, bietet die Verwendung der Funktion **INPUT$. Während **INPUT #** und **LINE INPUT #** Zeile für Zeile aus einer sequentiellen Datei lesen, liest **INPUT$** eine gegebene Anzahl von Zeichen aus einer Datei, wie in den folgenden Beispielen gezeigt:

Anweisung	***Funktion***
`X$ = INPUT$(100,#1)`	Liest 100 Zeichen aus Datei Nummer 1 und weist sie alle der Zeichenkettenvariablen `X$` zu.
`Test$ = INPUT$(1,#2)`	Liest ein Zeichen aus Datei Nummer 2 und weist es der Zeichenkettenvariablen `Test$` zu.

Die Funktion **INPUT$** ohne eine Dateinummer liest Eingaben immer von der Standardeingabe (normalerweise die Tastatur).

Die **INPUT$**-Funktion führt den als "Binäreingabe" bezeichneten Vorgang aus; das heißt, daß sie eine Datei als eine unformatierte Folge von Zeichen liest. Zum Beispiel erkennt sie eine Sequenz Wagenrücklauf-Zeilenvorschub nicht als Kennzeichen für das Ende einer Eingabeoperation. Daher ist **INPUT$** am besten für das Lesen jedes einzelnen Zeichens aus einer Datei, sowie für Binär- oder Nicht-ASCII-Dateien geeignet.

Beispiel

Das folgende Programm kopiert die genannte Binärdatei auf den Bildschirm, wobei es nur alphanumerische und Interpunktionszeichen im ASCII-Bereich von 32 bis 126, Tabulatoren, Wagenrückläufe und Zeilenvorschübe ausgibt:

```
' 9 ist der ASCII-Wert für horizontales Tab, 10 ist der
' ASCII-Wert für Zeilenvorschub und 13 ist der ASCII-Wert
' für Wagenrücklauf:
CONST ZEILVORSCH = 10, WAGENRUECK = 13, TABZEICH = 9

INPUT "Name der auszugebenden Datei: ", DateiName$
IF DateiName$ = "" THEN END

OPEN DateiName$ FOR INPUT AS #1

DO UNTIL EOF(1)
   Zeichen$ = INPUT$(1, #1)
   ZeichWert = ASC(Zeichen$)
   SELECT CASE ZeichWert
      CASE 32 TO 126
         PRINT Zeichen$;
      CASE TABZEICH, WAGENRUECK
         PRINT Zeichen$;
      CASE ZEILVORSCH
         IF AltZeichWert <> WAGENRUECK THEN PRINT Zeichen$
      CASE ELSE
         ' Dies ist keines der Zeichen, für die das Programm
         ' vorgesehen ist, also schreibe nichts.
    END SELECT
    AltZeichWert = ZeichWert
LOOP
```

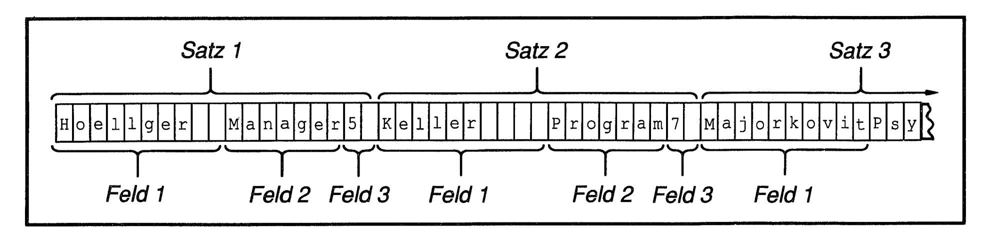
3.4.6 Die Verwendung von Direktzugriffsdateien

Dieser Abschnitt erläutert die Organisation von Datensätzen in Direktzugriffsdatendateien und zeigt anschließend, wie Daten aus einer Datei gelesen bzw. in eine für direkten Zugriff geöffnete Datei geschrieben werden können.

3.4.6.1 Datensätze in Direktzugriffsdateien

Datensätze für Direktzugriff werden anders als sequentielle Datensätze gespeichert. Jeder Datensatz für Direktzugriff ist, genau wie jedes Feld innerhalb des Datensatzes, mit einer festen Länge definiert. Diese festen Längen bestimmen den Anfang bzw. das Ende eines Datensatzes oder Feldes, da keine Felder trennende Kommata oder keine Sequenzen Wagenrücklauf-Zeilenvorschub zwischen den Datensätzen stehen. Abbildung 3.3 zeigt, wie drei Datensätze in einer Direktzugriffsdatei erscheinen können.

Abbildung 3.3 Datensätze in einer Direktzugriffsdatei



Beim Speichern von Datensätzen, die Zahlen enthalten, erweisen sich Direktzugriffsdateien sparsamer im Gebrauch von Speicherplatz auf der Diskette als sequentielle Dateien. Der Grund dafür ist, daß sequentielle Dateien Zahlen als eine Folge von ASCII-Zeichen, die je eine Ziffer darstellen, speichern, während Direktzugriffsdateien Zahlen in einem komprimierten Binärformat speichern.

Zum Beispiel wird die Zahl 17.000 in einer sequentiellen Datei mit 5 Bytes, die je einer Ziffer entsprechen, dargestellt. Wenn 17.000 jedoch als ein ganzzahliges Feld eines Datensatzes für Direktzugriff gespeichert wird, nimmt sie nur zwei Bytes auf der Diskette ein.

Im allgemeinen benötigen Ganzzahlen (Integer) in Direktzugriffsdateien zwei Bytes, lange Ganzzahlen (Long Integer) und Zahlen einfacher Genauigkeit (Single Precision) erfordern vier Bytes, und Zahlen doppelter Genauigkeit (Double Precision) acht Bytes.

3.4.6.2 Ergänzen einer Direktzugriffsdatei mit Daten

Ein Programm, das einer Direktzugriffsdatei Daten hinzufügt, ist wie folgt zu erstellen:

1. Definieren der Felder für jeden Datensatz.
2. Öffnen der Datei im Direktzugriffsmodus und Angabe der Länge jedes Datensatzes.
3. Eingabe für einen neuen Datensatz einlesen und den Datensatz in der Datei speichern.

Jeder dieser drei Schritte ist jetzt bedeutend leichter als in BASICA, wie aus nachstehendem Beispiel ersichtlich.

Definieren von Feldern

Sie können Ihren eigenen Datensatz (Record) anhand der Anweisung **TYPE...END TYPE** definieren, die die Erstellung eines zusammengesetzten Datentyps ermöglicht, der Zeichenketten und numerische Elemente mischt. Dies ist ein großer Vorteil gegenüber der früheren Methode, Datensätze mit einer **FIELD**-Anweisung einzurichten, die es erforderte, jedes Feld als Zeichenkette zu definieren. Durch die Definierung eines Datensatzes anhand von **TYPE...END TYPE** erübrigt sich die Notwendigkeit Funktionen zu verwenden, die numerische Daten in Zeichenketten (**MK***Typ***$**, **MKSMBF$** und **MKDMBF$**) und Zeichenketten in numerische Daten (**CV***Typ*, **CVSMBF** und **CVDMBF**) umwandeln.

Der folgende Ausschnitt stellt diese beiden Methoden zur Definition von Datensätzen gegenüber:

- **TYPE...END TYPE** definierter Datensatz:

```
' Definiere die Struktur von SatzTyp:
  TYPE SatzTyp
     VorName  AS STRING * 10
     NachName AS STRING * 15
     Alter    AS INTEGER
     Gehalt   AS SINGLE
  END TYPE
' Deklariere die Variable DatensatzVar mit dem Typ SatzTyp:
  DIM SatzVar AS SatzTyp
  .
  .
  .
```

- Mit **FIELD** definierter Datensatz:

```
' Definiere die Länge der Felder im temporären
' Speicherpuffer:
FIELD #1, 10 AS VorName$,15 AS NachName$, 2 AS Alter$, _
   4 AS Gehalt$
.
.
.
```

Öffnen der Datei und Angabe der Datensatzlänge

Da die Länge eines Datensatzes für Direktzugriff festgelegt ist, sollten Sie dem Programm die vorgesehene Datensatzlänge mitteilen; andernfalls ist die Datensatzlänge standardmäßig 128 Bytes.

Zur Angabe der Datensatzlänge ist die Klausel **LEN** = in der **OPEN**-Anweisung zu verwenden. Die nächsten beiden Ausschnitte, die die oben aufgeführten Gegenüberstellungen weiterführen, zeigen die Verwendung von **LEN** = :

- Angabe der Datensatzlänge für einen mit der Anweisung **TYPE...END TYPE** definierten Datensatz:

```
.
.
.
' Öffne die Direktzugriffsdatei und gib die Länge eines
' Datensatzes als gleich zu der Länge der Variablen
' SatzVar an:
OPEN "angest.dat" FOR RANDOM AS #1 LEN=LEN(SatzVar)
.
.
.
```

- Angabe der Datensatzlänge für einen mit **FIELD** definierten Datensatz:

```
.
.
.
' Öffne die Direktzugriffsdatei und gib die Länge eines
' Datensatzes an:
OPEN "angest.dat" FOR RANDOM AS #1 LEN = 31
.
.
.
```

Wie Sie feststellen können, müssen Sie bei der Verwendung von **FIELD** die Längen der Felder selbst addieren (`VorName$` hat 10 Bytes, `NachName$` hat 15 Bytes, `Alter$` hat 2 Bytes, `Gehalt$` hat 4 Bytes, so daß der Datensatz 10 + 15 + 2 + 4 = 31 Bytes lang ist). Mit **TYPE...END TYPE** sind diese Berechnungen nicht länger erforderlich. Stattdessen ist lediglich die Funktion **LEN** zur Ermittlung der Länge einer Variablen, die zur Aufnahme der Datensätze festgelegt wurde, zu benutzen (in diesem Fall `SatzVar`).

Eingabe von Daten in einen Datensatz und Speichern des Datensatzes

Sie können Daten direkt in die Elemente eines benutzerdefinierten Datensatzes eingeben, ohne sich um rechts- und linksbündige Ausrichtung der Eingaben innerhalb eines Feldes mit **LSET** oder **RSET** zu kümmern. Vergleichen Sie die folgenden zwei Ausschnitte, die die vorhergehenden Beispiele fortführen, um den durch dieses Verfahren eingesparten Programmcode festzustellen:

- Eingabe von Daten in einen Datensatz für Direktzugriff und Speichern des Datensatzes anhand von **TYPE...END TYPE**:

```
.
.
.
' Gib die Daten ein:
INPUT "Vorname";  SatzVar.Vorname
INPUT "Nachname"; SatzVar.NachName
INPUT "Alter";    SatzVar.Alter
INPUT "Gehalt";   SatzVar.Gehalt
' Speichere den Datensatz:
PUT #1, , SatzVar
.
.
.
```

- Eingabe von Daten in einen Datensatz für Direktzugriff und Speichern des Datensatzes mit **FIELD**:

```
.
.
.
' Gib die Daten ein:
INPUT "Vorname";  Vornm$
INPUT "Nachname"; NachNm$
INPUT "Alter";    AlterWert%
INPUT "Gehalt";   GehaltWert!
' Richte die Daten in den Speicherpuffer-Feldern
' linksbündig aus, und wandle Zahlen mit den Funktionen
' MKI$ und MKS$ in Dateizeichenketten um.
LSET VorName$     = VorNm$
LSET NachName$    = NachNm$
LSET Alter$       = MKI$(AlterWert%)
LSET Gehalt$      = MKS$(GehaltWert!)
'Speichere den Datensatz:
PUT #1
.
.
.
```

Zusammenfassung

Das folgende Programm faßt alle oben angegebenen Schritte – Definieren von Feldern, Angabe der Datensatzlänge, Eingabe von Daten und Speichern der eingegebenen Daten – zusammen, um eine als *bestand.dat* benannte Datendatei für Direktzugriff zu öffnen und ihr Datensätze hinzuzufügen:

```
DEFINT A-Z

' Definiere die Struktur eines einzelnen Datensatzes der
' Direktzugriffsdatei. Jeder Datensatz besteht aus vier
' Feldern fester Länge ("TeilNummer", "Beschreib",
' "EinzPreis", und "Menge"):
TYPE BestPosten
   TeilNummer  AS STRING *  6
   Beschreib   AS STRING * 20
   EinzPreis   AS SINGLE
   Menge       AS INTEGER
END TYPE

' Deklariere eine Variable (BestSatz), die den obigen Typ
' verwendet:
DIM BestSatz AS BestPosten

' Öffne die Direktzugriffsdatei, wobei die Länge eines
' Datensatzes als die Länge der Variablen BestSatz
' festgelegt wird:
OPEN "bestand.dat" FOR RANDOM AS #1 LEN=LEN(BestSatz)

' Verwende LOF(), um die Anzahl der sich bereits in der
' Datei befindenden Datensätze zu bestimmen, so daß neue
' Datensätze nach diesen angefügt werden:
SatzNummer = LOF(1) \ LEN(BestSatz)

' Füge nun neue Datensätze hinzu:
DO

   ' Gib Daten für einen Bestandsdatensatz per Tastatur ein
   ' und speichere die Daten in den unterschiedlichen
   ' Elementen der Variablen BestSatz:
   INPUT "Teilnummer ?   ",  BestSatz.TeilNummer
   INPUT "Beschreibung?  ",  BestSatz.Beschreib
   INPUT "Einzelpreis ?  ",  BestSatz.EinzPreis
   INPUT "Menge       ?  ",  BestSatz.Menge

   SatzNummer = SatzNummer + 1
```

```
    ' Schreibe die Daten von BestSatz in einen neuen
    ' Datensatz der Datei:
    PUT #1, SatzNummer, BestSatz

    ' Prüfe, ob weitere Daten gelesen werden sollen:
    INPUT "Weiter (J/N)? ", Antw$
LOOP UNTIL UCASE$(Antw$) = "N"

' Fertig. Schließe die Datei und beende:
CLOSE #1
END
```

Wenn die Datei *bestand.dat* bereits existiert, wird dieses Programm weitere Datensätze an die Datei anfügen, ohne die jeweiligen Datensätze, die sich bereits in der Datei befinden, zu überschreiben. Die folgende Schlüsselanweisung ermöglicht diesen Vorgang:

```
SatzNummer = LOF(1) \ LEN(BestSatz)
```

Es handelt sich um folgenden Vorgang:

1. Die Funktion `LOF(1)` berechnet die Gesamtzahl von Bytes in der Datei *bestand.dat*. Wenn *bestand.dat* neu ist oder keine Datensätze enthält, gibt `LOF(1)` den Wert `0` zurück.
2. Die Funktion `LEN(BestSatz)` berechnet die Anzahl von Bytes in einem Datensatz (`BestSatz` ist so definiert, daß sie dieselbe Struktur wie der benutzerdefinierte Typ `BestPosten` aufweist).
3. Daher entspricht die Satzanzahl dem Gesamtbetrag von Bytes in der Datei geteilt durch die Byteanzahl eines Datensatzes.

Dies ist ein weiterer Vorteil der Verwendung eines Datensatzes fester Länge: Da jeder Datensatz dieselbe Größe hat, kann obige Formel zur Berechnung der Datensatzanzahl in der Datei angewandt werden. Dies funktioniert offensichtlich nicht bei sequentiellen Dateien, da diese unterschiedliche Längen aufweisen.

3.4.6.3 Sequentielles Lesen von Daten

Mit der in dem vorhergehenden Abschnitt beschriebenen Vorgehensweise zur Berechnung der Datensatzanzahl in einer Direktzugriffsdatei läßt sich ein Programm schreiben, das alle Datensätze dieser Datei liest.

Beispiel

Das folgende Programm liest sequentiell Sätze (vom ersten bis zum letzten gespeicherten Datensatz) aus der im vorhergehenden Beispiel erstellten Datei *bestand.dat*:

```
' Definiere eine Datensatzstruktur (Typ) für Datensätze
' einer Direktzugriffsdatei:
TYPE BestPosten
   TeilNummer  AS STRING *  6
   Beschreib   AS STRING * 20
   EinzPreis   AS SINGLE
   Menge       AS INTEGER
END TYPE
' Deklariere eine Variable (BestSatz), die den obigen Typ
' verwendet:
DIM BestSatz AS BestPosten
'Öffne die Direktzugriffsdatei.
OPEN "bestand.dat" FOR RANDOM AS #1 LEN = LEN(BestSatz)
' Berechne die Anzahl der Datensätze in der Datei:
AnzVonSaetzen = LOF(1) \ LEN(BestSatz)
' Lies die Datensätze und schreibe die Daten auf den
' Bildschirm:
FOR SatzNummer = 1 TO AnzVonSaetzen
   ' Lies die Daten eines neuen Datensatzes in der Datei:
   GET #1, SatzNummer, BestSatz
   ' Gib die Daten auf den Bildschirm aus:
   PRINT "Teilnummer   : ", BestSatz.TeilNummer
   PRINT "Beschreibung : ", BestSatz.Beschreib
   PRINT "Einzelpreis  : ", BestSatz.EinzPreis
   PRINT "Menge        : ", BestSatz.Menge
NEXT
' Fertig; schließe die Datei und beende.
CLOSE #1
END
```

Es ist nicht erforderlich, *bestand.dat* vor Lesen der darin enthaltenen Informationen zu schließen. Das Öffnen einer Datei für Direktzugriff erlaubt das Schreiben und Lesen in der Datei mit einer einzigen **OPEN**-Anweisung.

3.4.6.4 Zugriff auf bestimmte Direktzugriffsdateien anhand von Datensatznummern

Jeder Datensatz einer Direktzugriffsdatei kann nach Angabe der Datensatznummer in einer **GET**-Anweisung gelesen werden. In jeden Datensatz einer Direktzugriffsdatei kann nach Angabe der Datensatznummer in einer **PUT**-Anweisung geschrieben werden. Dies ist einer der Hauptvorteile der Direktzugriffsdateien gegenüber sequentiellen Dateien, da sequentielle Dateien keinen direkten Zugriff auf einen bestimmten Datensatz erlauben.

Das in Abschnitt 3.6.2 aufgeführte Anwendungsbeispiel, *index.bas*, verdeutlicht eine Methode, einen bestimmten Datensatz sehr schnell zu finden, indem ein Index der Datensatznummern durchsucht wird.

Beispiel

Der folgende Ausschnitt zeigt, wie **GET** mit einer Datensatznummer zu verwenden ist:

```
DEFINT A-Z          ' Standarddatentyp ist Ganzzahl.
CONST FALSCH = 0, WAHR = NOT FALSCH
TYPE BestPosten
   TeilNummer  AS STRING *  6
   Beschreib   AS STRING * 20
   EinzPreis   AS SINGLE
   Menge       AS INTEGER
END TYPE
DIM BestSatz AS BestPosten
OPEN "bestand.dat" FOR RANDOM AS #1 LEN=LEN(BestSatz)
AnzVonSaetzen= LOF(1) \ LEN(BestSatz)
WeitereSaetze = WAHR
DO
   PRINT "Geben Sie die Satznummer des gewünschten ";
   PRINT "Teiles ein: ";
   PRINT "(0 zum Beenden): ";
   INPUT "", SatzNummer
```

```
   IF SatzNummer > 0 AND SatzNummer < AnzVonSaetzen    THEN
      ' Lies den Satz, dessen Nummer eingegeben wurde
      ' und speichere ihn in BestSatz:
      GET #1, SatzNummer, BestSatz
      ' Zeige den Satz an:
      .
      .
      .
   ELSEIF SatzNummer = 0 THEN
      WeitereSaetze = FALSCH
   ELSE
      PRINT "Eingegebener Wert außerhalb des Bereichs."
   END IF
LOOP WHILE WeitereSaetze
END
```

3.4.7 Binärdatei-E/A

Der binäre Zugriff stellt zusätzlich zum Direktzugriff und dem sequentiellen Zugriff die dritte Möglichkeit dar, Daten einer Datei zu lesen oder zu schreiben. Eine Datei für Binär-E/A ist anhand folgender Anweisung zu öffnen:

OPEN *Datei* **FOR BINARY AS** *#Dateinummer*

Der Binärzugriff ist eine Möglichkeit, an die reinen Bytes einer beliebigen Datei, nicht nur einer ASCII-Datei, heranzukommen. Daher ist der Binärzugriff besonders nützlich beim Lesen oder Verändern von Dateien, die in einem Nicht-ASCII-Format gespeichert sind, wie z. B. Microsoft Word -Dateien oder ausführbare (*.exe*) Dateien.

Im Binärmodus geöffnete Dateien werden als eine unformatierte Folge von Bytes behandelt. Obwohl ein Datensatz (eine mit einem benutzerdefinierten Typ deklarierte Variable) aus einer im Binärmodus geöffneten Datei gelesen bzw. geschreiben werden kann, darf die Datei selbst nicht aus Datensätzen fester Länge aufgebaut sein. Tatsächlich muß Binär-E/A gar nicht mit Datensätzen umgehen, es sei denn, Sie stellen sich jedes Byte in einer Datei als einen separaten Datensatz vor.

3.4.7.1 Vergleich zwischen Binärzugriff und Direktzugriff

Sowohl im Modus Binärzugriff als auch im Modus Direktzugriff können Sie nach einer einzigen **OPEN**-Anweisung aus einer Datei lesen und in eine Datei schreiben. (Der binäre Zugriff unterscheidet sich ebenfalls vom sequentiellen Zugriff, bei dem vor Wechsel zwischen Lesen und Schreiben eine Datei zuerst geschlossen und anschließend neu geöffnet werden muß.) Außerdem können Sie sich mit Binärzugriff, wie mit Direktzugriff, innerhalb einer geöffneten Datei rückwärts und vorwärts bewegen. Binärzugriff unterstützt sogar dieselben Anweisungen, die zum Lesen und Schreiben von Direktzugriffsdateien verwendet werden:

{**GET** | **PUT**} **[#]** *Dateinummer*, [*Position*], *Variable*

Variable kann hier jeden Typ haben, einschließlich einer Zeichenkette variabler Länge oder eines benutzerdefinierten Typs, und Position zeigt auf eine Stelle in der Datei, an die die nächste **GET**- oder **PUT**-Anweisung ausgeführt wird. (Die Position ist relativ zum Beginn der Datei; das heißt, das erste Byte in der Datei hat Position eins, das zweite Byte Position zwei usw.) Wird das Argument Position ausgelassen, bewegen aufeinanderfolgende **GET**- bzw. **PUT**-Anweisungen den Dateizeiger sequentiell vom ersten bis zum letzten Byte der Datei.

Die **GET**-Anweisung liest eine Anzahl von Bytes aus der Datei, die gleich der Länge von *Variable* ist. Ähnlich schreibt die **PUT**-Anweisung eine Anzahl von Bytes, die gleich der Länge von *Variable* ist, in die Datei. Wenn *Variable* zum Beispiel den Typ Ganzzahl hat, dann liest **GET** zwei Bytes in *Variable*; wenn sie den Typ einfache Genauigkeit hat, liest **GET** vier Bytes. Wird demnach kein Argument *Position* in einer **GET**- oder **PUT**-Anweisung angegeben, wird der Dateizeiger um einen Abstand gleich der Länge von *Variable* weiterbewegt.

Die Anweisung **GET** und die Funktion **INPUT$** sind die einzigen Möglichkeiten, Daten aus einer im Binärmodus geöffneten Datei zu lesen. Die Anweisung **PUT** ist die einzige Möglichkeit, Daten in eine im Binärmodus geöffnete Datei zu schreiben.

Zum Unterschied von Direktzugriff ermöglicht Binärzugriff den Zugriff auf jede Byte-Position einer Datei und anschließend das Lesen und Schreiben der gewünschten Anzahl von Bytes. Im Gegensatz dazu sind Sie bei Direktzugriff gezwungen, sich jedesmal zu einer Datensatzgrenze zu bewegen und eine festgelegte Anzahl von Bytes (die Länge eines Datensatzes) zu lesen.

3.4.7.2 Positionierung des Dateizeigers mit Hilfe von SEEK

Wenn Sie den Dateizeiger ohne gleichzeitige Anwendung irgendeiner E/A auf eine bestimmte Stelle in der Datei bewegen wollen, ist die **SEEK**-Anweisung zu benutzen:

SEEK *Dateinummer, Position*

Nach einer **SEEK**-Anweisung beginnt die nächste Lese- oder Schreiboperation in der mit *Dateinummer* geöffneten Datei mit dem in *Position* festgehaltenen Byte.

Das Gegenstück zur **SEEK**-Anweisung ist die Funktion **SEEK** mit der Syntax:

SEEK (*Dateinummer*)

Diese Funktion zeigt die Byte-Position an, wo die nächste Lese- oder Schreiboperation beginnt. (Beim Zugriff auf eine Datei anhand von Binär-E/A ergeben die Funktionen **LOC** und **SEEK** ähnliche Ergebnisse, mit dem Unterschied, daß **LOC** die Position des letzten gelesenen oder geschriebenen Bytes angibt, während **SEEK** die Position des nächsten zu lesenden oder zu schreibenden Bytes angibt.)

Die **SEEK**-Anweisung und -Funktion arbeiten ebenfalls mit Dateien, die für sequentiellen oder direkten Zugriff geöffnet sind. Bei sequentiellem Zugriff verhalten sich sowohl die Anweisung als auch die Funktion genau wie bei binärem Zugriff; das heißt, die Anweisung **SEEK** bewegt den Dateizeiger auf bestimmte Byte-Positionen und die Funktion **SEEK** liefert Informationen über das nächste zu lesende oder zu schreibende Byte.

Wird eine Datei jedoch für direkten Zugriff geöffnet, kann die **SEEK**-Anweisung den Dateizeiger jedoch nur zum Beginn eines Datensatzes bewegen, nicht aber zu einem Byte innerhalb des Datensatzes. Ebenso gibt die Funktion **SEEK** lediglich die Nummer des nächsten Datensatzes zurück und nicht die Position des nächsten Bytes.

Beispiel

Das folgende Programm öffnet eine QuickBASIC Quick-Bibliothek; anschließend liest und schreibt es die Namen von BASIC-Prozeduren und anderer externer Symbole, die die Bibliothek enthält. Dieses Programm befindet sich in der Datei *qbibdru.bas* auf den QuickBASIC Original-Disketten.

```
' Dieses Programm gibt die Namen von Quickbibliotheks-
' Prozeduren aus
DECLARE SUB DruckSym (SymbolBeg AS INTEGER, QKopfPos AS
LONG)

TYPE ExeKopf                    ' Teil des DOS .exe-Kopfes
                                ' (header)
    andere1 AS STRING * 8       ' Andere Kopfinformation
    ParKopf AS INTEGER          ' Größe des Kopfes in
                                ' Paragraphen
    andere2 AS STRING * 10      ' Andere Kopfinformation
    IP      AS INTEGER          ' Startwert von IP
    CS      AS INTEGER          ' Startwert (relativ) von
END TYPE                        ' CS
```

```
TYPE QBKopf                   ' QLB-Kopf
    QBUeber   AS STRING * 6 ' QB-spezifische
                             ' Überschrift
    Magie     AS INTEGER     ' Magisches Wort:
                             ' identifiziert Datei als
                             ' Quick-Bibliothek
    SymbolBeg AS INTEGER     ' Offset vom Kopf zum
                             ' ersten Codesymbol
    DatenBeg AS INTEGER      ' Offset vom Kopf zum
                             ' ersten Datensymbol
END TYPE
TYPE QbSym                   ' Symboleintrag in Quick-
                             ' Bibliothek
    Flags    AS INTEGER        ' Symbolkennzeichen
    NameBeg  AS INTEGER      ' Offset zur Namentabelle
    andere   AS STRING * 4   ' Andere Kopfinformation
END TYPE
DIM EKopf AS ExeKopf, QKopf AS QBKopf, QKopfPos AS LONG
PRINT "Geben Sie den Namen der Quick-Bibliothek ein";
INPUT ":", DateiName$
DateiName$ = UCASE$(DateiName$)
IF INSTR(DateiName$,".qlb")=0 THEN
    DateiName$=DateiName$ + ".qlb"
ENDIF
PRINT "Geben Sie den Namen der Ausgabedatei ein, oder"
PRINT "druecken Sie <EINGABETASTE> zur Ausgabe auf den"
PRINT "Bildschirm";
INPUT ":", AusgDatei$
AusgDatei$ = UCASE$(AusgDatei$)
IF AusgDatei$ = "" THEN AusgDatei$ = "CON"
OPEN DateiName$ FOR BINARY AS #1
OPEN AusgDatei$ FOR OUTPUT AS #2
GET #1, , EKopf            ' Lies den EXE-Formatkopf.
QKopfPos = (EKopf.ParKopf + EKopf.CS) * 16 + EKopf.IP    + 1
GET #1, QKopfPos, QKopf    ' Lies den Quicklib-
                           ' Formatkopf.
IF QKopf.Magie <> &H6C75 THEN
   PRINT "Keine Quick- Bibliothek!" : END
ENDIF
PRINT #2, "Codesymbole:":PRINT #2
DruckSym QKopf.SymbolBeg, QKopfPos    ' Drucke Codesymbole
```

```
PRINT #2,
PRINT #2, "Datensymbole:":PRINT #2, ""
DruckSym QKopf.DatenBeg, QKopfPos     ' Drucke Datensymbole

SUB DruckSym (SymbolBeg AS INTEGER, QKopfPos AS LONG)
   DIM QlbSym AS QbSym
   DIM NaechstSym AS LONG, AktSym AS LONG

   ' Berechne die Lage des ersten Symboleintrags,
   ' lies dann diesen Eintrag:
   NaechstSym = QKopfPos + QKopf.SymbolBeg
   GET #1, NaechstSym, QlbSym
   DO
      NaechstSym = SEEK(1)      ' Sichere Lage des
                                ' nächsten Symbols.
      AktSym = QKopfPos + QlbSym.NameBeg
      SEEK #1, AktSym           ' Verwende SEEK zur
                                ' Bewegung auf den Namen
                                ' des aktuellen
                                ' Symboleintrags.
      Erwart$ = INPUT$(40, 1)   ' Lies die längste legale
                                ' Zeichenkette plus ein
                                ' zusätzliches Byte für
                                ' das abschließende Null-
                                ' Zeichen (CHR$(0)).
      ' Ziehe den mit Null beendeten Namen heraus:
      SName$ = LEFT$(Erwart$, INSTR(Erwart$, CHR$(0)))

      ' Gib nur solche Namen aus, die nicht mit "__",
      ' "$" oder "b$" beginnen, weil diese Namen
      ' normalerweise als
      ' reserviert angesehen werden:
      T$ = LEFT$(SName$, 2)
      IF T$ <> "__" AND LEFT$(SName$, 1) <> "$" AND_
      UCASE$(T$) <> "B$" THEN
         PRINT #2, " " + SName$
      END IF

      GET #1, NaechstSym, QlbSym  ' Lies einen
                                  ' Symboleintrag.
   LOOP WHILE QlbSym.Flags        ' Flags=0 (falsch)
                                  ' bedeutet Ende der
                                  ' Tabelle.

END SUB
```

3.5 Die Arbeit mit den Geräten

Microsoft-BASIC unterstützt Geräte-E/A. Das heißt, daß bestimmte Peripherieanschlüsse des Rechners wie Datendateien auf Diskette für E/A geöffnet werden können. Eingabe von diesen Geräten oder Ausgabe auf diese Geräte (Device) kann mit den in Tabelle 9.4, "In der Datendatei-E/A verwendete Anweisungen und Funktionen ", aufgeführten Anweisungen vorgenommen werden, mit den Ausnahmen, die in Abschnitt 3.5.1, "Unterschiede zwischen Geräte- und Datei-E/A" gezeigt werden. Tabelle 3.1 listet die von BASIC unterstützten Geräte.

Tabelle 3.1 Für E/A von BASIC unterstützte Geräte

Name	*Gerät*	*Unterstützter E/A-Modus*
COM1:	erster serieller Anschluß	Ein- und Ausgabe
COM2:	zweiter serieller Anschluß	Ein- und Ausgabe
CONS:	Bildschirm	nur Ausgabe
KYBD:	Tastatur	nur Eingabe
LPT1:	erster Drucker	nur Ausgabe
LPT2:	zweiter Drucker	nur Ausgabe
LPT3:	dritter Drucker	nur Ausgabe
SCRN:	Bildschirm	nur Ausgabe

3.5.1 Unterschiede zwischen Geräte- und Datei-E/A

Bestimmte für Datei-E/A verwendete Funktionen und Anweisungen sind für Geräte-E/A nicht erlaubt, während andere Anweisungen und Funktionen sich bei Geräten anders verhalten. Im folgenden werden einige dieser Unterschiede aufgeführt:

- Die Geräte CONS:, SCRN: und LPT*n*: können nicht in den Eingabe- oder Anfügemodi geöffnet werden.
- Das Gerät KYBD: kann nicht im Ausgabe- oder Anfügemodus geöffnet werden.
- Die Funktionen **EOF**, **LOC** und **LOF** können nicht mit den Geräten CONS:, KYBD:, LPT*n*: oder SCRN: verwendet werden.
- Die Funktionen **EOF**, **LOC** und **LOF** können mit dem seriellen Gerät COM*n*: verwendet werden; die Werte, die diese Funktionen zurückgeben, haben jedoch eine andere Bedeutung als die Werte, die sie bei Verwendung mit Datendateien zurückgeben. (In Abschnitt 3.5.2 finden Sie eine Erklärung zur Wirkungsweise dieser Funktionen mit COM*n*:.)

Beispiel

Das folgende Programm zeigt, wie die Geräte LPT1: oder SCRN: mit derselben Syntax, die für Datendateien gilt, für Ausgabe geöffnet werden können. Dieses Programm liest alle Zeilen aus der vom Benutzer gewählten Datei und gibt diese Zeilen anschließend je nach der Wahl des Benutzers auf den Bildschirm oder auf den Drucker aus.

```
CLS
' Gib den Namen der anzuzeigenden Datei ein:
INPUT "Name der anzuzeigenden Datei: ", DateiNam$

' Wenn kein Name eingegeben wurde, beende das Programm,
' andernfalls öffne die angegebene Datei zum Lesen (INPUT):
IF DateiNam$ = "" THEN
    END
ELSE OPEN DateiNam$ FOR INPUT AS #1
END IF

' Gib Wahl bei Ausgabegerät der Datei an (Bildschirm oder
' Drucker); wiederhole Eingabeaufforderung solange, bis
' entweder die Tasten "B" oder "D" betätigt werden:
DO
   ' Gehe zu Zeile 2, Spalte 1 auf dem Bildschirm und
   ' schreibe Anfrage:
   LOCATE 2, 1, 1
   PRINT "Sende Ausgabe zum Bildschirm (B)";
   PRINT "oder zum Drucker (D): ";

   ' Überschreibe alles nach der Anfrage:
   PRINT SPACE$(2);

   ' Positioniere den Cursor hinter der Anfrage und mache
   ' ihn sichtbar:
   LOCATE 2, 47, 1
   Wahl$ = UCASE$(INPUT$(1))      ' Lies Eingabe.
   PRINT Wahl$
LOOP WHILE Wahl$ <> "B" AND Wahl$ <> "D"

' Abhängig von der betätigten Taste den Drucker oder den
' Bildschirm für Ausgabe öffnen:
SELECT CASE Wahl$
   CASE "D"
      OPEN "LPT1:" FOR OUTPUT AS #2
      PRINT "Ausgabe der Datei auf Drucker."
   CASE "B"
      CLS
      OPEN "SCRN:" FOR OUTPUT AS #2
END SELECT
```

```
' Setze die Breite des gewählten Ausgabegerätes auf 80
' Spalten:
WIDTH #2, 80

' Solange sich in der Datei Zeilen befinden, lies eine Zeile
' aus der Datei und gib sie auf das gewählte Gerät aus:
DO UNTIL EOF(1)
   LINE INPUT #1, ZeilPuffer$
   PRINT #2, ZeilPuffer$
LOOP

CLOSE          ' Beende Eingabe aus der Datei und Ausgabe auf
               ' das Gerät.
END
```

3.5.2 Datenübertragung über einen seriellen Anschluß

Die Anweisung **OPEN** "**COM***n***:**" (wobei *n* 1 oder 2 - falls zwei serielle Anschlüsse vorhanden sind - sein kann) erlaubt es Ihnen, den/die seriellen Anschluß/Anschlüsse des Computers für serielle (Bit für Bit) Datenübertragung mit anderen Computern oder mit Peripheriegeräten wie Modems oder seriellen Druckern zu öffnen. Im folgenden werden nur einige anzugebende Parameter genannt.

- Geschwindigkeit der Datenübertragung, gemessen in "Baud" (Bits pro Sekunde).
- Ob und wie Übertragungsfehler festgestellt werden sollen.
- Anzahl der zu verwendenden Stopbits (1, 1,5 oder 2), um das Ende eines übertragenen Bytes anzuzeigen.
- Anzahl der Bits, die eigentliche Daten in jedem übertragenen oder empfangenen Datenbyte bilden.

Wenn der serielle Anschluß für Datenübertragung mit Peripheriegeräten und anderen Computern geöffnet wird, wird ein Eingabepuffer eingerichtet, der die von dem anderen Gerät gelesenen Bytes aufnimmt. Der Grund dafür ist die Tatsache, daß bei hohen Baud-Raten Zeichen schneller ankommen, als sie verarbeitet werden können. Die Standardgröße für diesen Puffer beträgt 512 Bytes und kann mit der Option **LEN** = *AnzBytes* der Anweisung **OPEN** "**COM***n***:**" verändert werden. Die von den Funktionen **EOF**, **LOC** und **LOF** angegebenen Werte liefern bei Verwendung mit einem Datenübertragungsgerät Informationen über die Größe dieses Puffers, wie in der folgenden Liste dargestellt.

Funktion	***Angegebene Information***
EOF	Ob Zeichen darauf warten, aus dem Eingabepuffer gelesen zu werden.
LOC	Die Anzahl der im Eingabepuffer wartenden Zeichen.
LOF	Der Betrag des im Eingabepuffer verbleibenden Platzes (in Bytes).

Da jedes Zeichen bei einer Datenübertragung wichtig sein kann, haben sowohl **INPUT#** als auch **LINE INPUT #** erhebliche Nachteile beim Lesen von Eingaben anderer Geräte. Dies liegt daran, daß **INPUT #** das Lesen von Daten in eine Variable unterbricht, wenn es ein Komma oder eine neue Zeile (und manchmal auch ein Leerzeichen oder ein doppeltes Anführungszeichen) antrifft, und **LINE INPUT #** das Lesen von Daten unterbricht, wenn es eine neue Zeile antrifft. Demnach ist **INPUT$** die am besten geeignete Funktion zur Verwendung bei Eingaben von einem Datenübertragungsgerät, da sie alle Zeichen liest.

Die folgende Zeile verwendet die Funktion **LOC**, um den Eingabepuffer auf die Anzahl der dort aus dem als Datei #1 geöffneten Datenübertragungsgerät wartenden Zeichen zu überprüfen. Die Zeile verwendet danach zum Lesen dieser Zeichen die Funktion **INPUT$**, die die Zeichen einer mit `ModemEingabe$` bezeichneten Variablen zuweist:

```
ModemEingabe$ = INPUT$(LOC(1), #1)
```

3.6 Anwendungsbeispiele

Die in diesem Abschnitt aufgeführten Anwendungsbeispiele umfassen ein bildschirmverwaltendes Programm, das einen Kalender für jeden Monat in jedem Jahr von 1899 bis 2099 ausgibt, ein Datei-E/A-Programm, das einen Index der Datensatznummern einer Direktzugriffsdatei erstellt und durchsucht, und ein Datenübertragungsprogramm, das den PC dazu veranlaßt, sich nach Anschluß an ein Modem wie eine Datenstation (Terminal) zu verhalten.

3.6.1 Immerwährender Kalender (*kal.bas*)

Nach der Aufforderung an den Benutzer, einen Monat von 1 bis 12 und ein Jahr von 1899 bis 2099 einzugeben, gibt das folgende Programm den Kalender für den gegebenen Monat und das angegebene Jahr aus. Die Prozedur `IstSchaltJahr` führt eine geeignete Anpassung des Kalenders für Monate in einem Schaltjahr durch.

Verwendete Anweisungen und Funktionen

Dieses Programm demonstriert die folgenden bildschirmverwaltenden Funktionen und Anweisungen, die in Abschnitt 3.1 bis 3.3 erläutert wurden:

- **INPUT**
- **INPUT$**
- **LOCATE**
- **POS(0)**
- **PRINT**
- **PRINT USING**
- **TAB**

Programm-Listing

Untenstehend wird das immerwährende Kalenderprogramm *kal.bas* aufgelistet:

```
DEFINT A-Z     ' Standardvariablentyp ist Ganzzahl.
' Definiere einen Datentyp für die Namen der Monate
' und die Anzahl von Tagen in jedem Monat:
TYPE MonatTyp
   Anzahl AS INTEGER    ' Anzahl von Tagen in dem Monat
   MName AS STRING * 9 ' Name des Monats
END TYPE
' Deklariere die verwendeten Prozeduren:
DECLARE FUNCTION IstSchaltJahr% (N%)
DECLARE FUNCTION HolEingabe% (Anfrage$, Zeile%,
NiedWert%, HoherWert%)
DECLARE SUB Schreibkalender (Jahr%, Monat%)
DECLARE SUB BerechnMonat (Jahr%, Monat%, StartTag%,
TotalTage%)
DIM MonatDatum(1 TO 12) AS MonatTyp
' Initialisiere Monatsdefinitionen mit den DATA-
' Anweisungen weiter unten.
FOR I = 1 TO 12
   READ MonatDatum(I).MName, MonatDatum(I).Anzahl
NEXT
' Hauptschleife, wiederhole für soviele Monate wie
' gewünscht:
DO
   CLS
```

```
' Hole Jahr und Monat als Eingabe:
   Jahr = HolEingabe("Jahr (1899 to 2099): ",1,1899,2099)
   Monat = HolEingabe("Monat (1 to 12): ",2,1,12)
   ' Gib den Kalender aus:
   SchreibKalender Jahr, Monat
   ' Ein weiteres Datum?
   LOCATE 13, 1              ' Plaziere Cursor in Zeile
                             ' 13, Spalte 1.
   PRINT "Neues Datum? ";    ' Halte Cursor auf
                             ' derselben Zeile.
   LOCATE , , 1, 0, 13       ' Schalte Cursor ein und
                             ' laß ihn ein Zeichen hoch
                             ' werden.
   Antw$ = INPUT$(1)         ' Warte auf Betätigen einer
                             ' Taste.
   PRINT Antw$               ' Gib betätigte Taste aus.
LOOP WHILE UCASE$(Antw$) = "J"
END
' Daten für die Monate eines Jahres:
DATA Januar, 31, Februar, 28, Maerz, 31
DATA April, 30, Mai, 31, Juni, 30, Juli, 31, August,    31
DATA September, 30, Oktober, 31, November, 30, Dezember, 31
'
' ===================== BERECHNMONAT ======================
'     Berechne den ersten Tag und die Anzahl der Tage
'     eines Monats
' =========================================================
'
SUB BerechnMonat (Jahr, Monat, StartTag, TotalTage) STATIC
   SHARED MonatDatum() AS MonatTyp
   CONST SCHALT = 366 MOD 7
   CONST NORMAL = 365 MOD 7
   ' Berechne Gesamtzahl der Tage (AnzTage) seit
   ' 1.1.1899.
   ' Beginne mit den ganzen Jahren:
   AnzTage = 0
   FOR I = 1899 TO Jahr - 1
      IF IstSchaltJahr(I) THEN          'Wenn Schaltjahr,
        AnzTage = AnzTage + SCHALT      'addiere 366 MOD 7.
      ELSE                              'Wenn normales
        AnzTage = AnzTage + NORMAL      'Jahr, addiere
      END IF                            '365 MOD 7.
   NEXT
```

```
   ' Addiere nun die Tage der ganzen Monate:
   FOR I = 1 TO Monat - 1
      AnzTage = AnzTage + MonatDatum(I).Anzahl
   NEXT
   ' Setze die Anzahl von Tagen in dem gewünschten
   ' Monat:
   TotalTage = MonatDatum(Monat).Anzahl
   ' Gleiche aus, wenn gewünschtes Jahr ein Schaltjahr
   ' ist:
   IF IstSchaltJahr(Jahr) THEN
      ' Wenn nach Februar, addiere eins zu den
      ' gesamten Tagen:
      IF Monat > 2 THEN
         AnzTage = AnzTage + 1
      ' Wenn Februar, addiere eins zu den Tagen des
      ' Monats:
      ELSEIF Monat = 2 THEN
         TotalTage = TotalTage + 1
      END IF
   END IF

   ' Der 1.1.1899 war ein Sonntag, daher ergibt die
   ' Berechnung von "AnzTage MOD 7" den Wochentag
   ' (Sonntag = 0, Montag = 1, Dienstag = 2, und so
   ' weiter) des ersten Tages für den eingegebenen
   ' Monat:
   StartTag = AnzTage MOD 7
END SUB
'
' =================== HOLEINGABE =====================
'     Fordert zur Eingabe auf und testet dann auf
'     gültigen Bereich
' ====================================================
'
FUNCTION HolEingabe (Anfrage$, Zeile, NiedWert,
HoherWert) STATIC

   ' Positioniere Anfrage in der angegebenen Zeile,
   ' schalte den Cursor ein und laß ihn ein Zeichen
   ' hoch werden:
   LOCATE Zeile, 1, 1, 0, 13
   PRINT Anfrage$;

   ' Sichere Spaltenposition:
   Spalte = POS(0)
```

```
   ' Gib Wert ein, bis er innerhalb des Bereiches
   ' liegt:
   DO
      LOCATE Zeile, Spalte    ' Positioniere Cursor an
                              ' das Ende der Anfrage.
      PRINT SPACE$(10)        ' Lösche alles bereits
                              ' vorhandene.
      LOCATE Zeile, Spalte    ' Positioniere Cursor
                              ' hinter Anfrage.
      INPUT "", Wert          ' Gib Wert ohne Anfrage
                              ' ein.
   LOOP WHILE (Wert < NiedWert OR Wert > HoherWert)

   ' Gib gültige Eingabe als Wert der Funktion zurück:
   HolEingabe = Wert

END FUNCTION
'
' ====================== ISTSCHALTJAHR =====================
' Bestimmt, ob ein Jahr ein Schaltjahr ist oder nicht
' ==========================================================
'

FUNCTION IstSchaltJahr (N)    STATIC
   ' Wenn das Jahr ohne Rest durch vier aber nicht
   ' durch 100 teilbar ist, oder wenn das Jahr ohne
   ' Rest durch 400 teilbar ist, dann ist es ein
   ' Schaltjahr:
   IstSchaltJahr = (N MOD 4 = 0 AND N MOD 100<>0) OR (N MOD
400=0)
END FUNCTION
'
' ==================== SCHREIBKALENDER =====================
'   Gibt einen formatierten Kalender für den Monat
'   in dem gegebenen Jahr aus
' ==========================================================
'
SUB SchreibKalender (Jahr, Monat)   STATIC
SHARED MonatDatum() AS MonatTyp

   ' Berechne den ersten Tag (So Mo Di ...) und die
   'Anzahl der Tage des Monats:
   BerechnMonat Jahr, Monat, StartTag, TotalTage
   CLS
   Kopf$ = RTRIM$(MonatDatum(Monat).MName)+ "," + STR$(Jahr)
```

```
' Berechne Stelle zur Zentrierung von Monat und
   ' Jahr:
   LinkRand = (35 - LEN(Kopf$)) \ 2

   ' Gib Kopf aus:
   PRINT TAB(LinkRand); Kopf$
   PRINT
   PRINT "So   Mo   Di   Mi   Do   Fr    Sa"
   PRINT

   ' Berechne erneut und gib Tab zum ersten Tag des
   ' Monats (So Mo Di ...) aus:
   LinkRand = 5 * StartTag + 1
   PRINT TAB(LinkRand);

   ' Gib die Tage des Monats aus:
   FOR I = 1 TO TotalTage
      PRINT USING "##   "; I;

      ' Springe auf nächste Zeile, wenn Cursor sich
      ' hinter Spalte 32 befindet:
      IF POS(0) > 32 THEN PRINT
   NEXT

END SUB
```

Ausgabe

```
          Februar, 1988

 So   Mo   Di   Mi   Do   Fr    Sa

       1    2    3    4    5    6
  7    8    9   10   11   12   13
 14   15   16   17   18   19   20
 21   22   23   24   25   26   27
 28   29


 Neues Datum?
```

3.6.2 Indexieren einer Direktzugriffsdatei (*index.bas*)

Das folgende Programm verwendet eine Indizierungstechnik zum Speichern und Wiederfinden von Datensätzen in einer Direktzugriffsdatei. Jedes Element des Datenfeldes `Index` verfügt über zwei Teile: ein Zeichenkettenfeld (`TeilNummer`) und ein Ganzzahlfeld (`SatzNummer`). Dieses Datenfeld ist im Feld `TeilNummer` alphabetisch sortiert, wodurch ein schnelles Suchen nach einer bestimmten Teilnummer im Datenfeld mit einem binären Suchalgorithmus ermöglicht wird.

Das Datenfeld `Index` funktioniert ähnlich wie das Stichwortverzeichnis eines Buches. Wenn Sie in einem Buch die Seiten finden möchten, die sich mit einem bestimmten Thema befassen, suchen Sie nach einem Eintrag zu diesem Thema im Stichwortverzeichnis. Der Eintrag weist auf eine Seitenzahl im Buch hin. Ähnlich sucht dieses Programm eine Teilnummer in dem alphabetisch sortierten Datenfeld `Index`. Wenn es die Teilnummer gefunden hat, zeigt die entsprechende Datensatznummer in dem Feld `SatzNummer` auf den Datensatz, der alle Informationen zu diesem Teil enthält.

Verwendete Anweisungen und Funktionen

Dieses Programm stellt folgende beim Zugriff auf Direktzugriffdateien verwendete Funktionen und Anweisungen dar:

- **TYPE...END TYPE**
- **OPEN...FOR RANDOM**
- **GET #**
- **PUT #**
- **LOF**

Programm-Listing

Untenstehend wird das Indizierungsprogramm für Direktzugriffsdateien *index.bas* aufgelistet:

```
DEFINT A-Z

' Definiere die symbolischen Konstanten, die in dem
' Programm global verwendet werden:
CONST FALSCH = 0, WAHR = NOT FALSCH

' Definiere eine Satzstruktur für Datensätze einer
' Direktzugriffsdatei:
TYPE BestPosten
```

```
   TeilNummer AS STRING * 6
   Beschreib  AS STRING * 20
   EinzPreis  AS SINGLE
   Menge      AS INTEGER
END TYPE

' Definiere eine Satzstruktur für jedes Element des
' Index:
TYPE IndexTyp
   SatzNummer AS INTEGER
   TeilNummer AS STRING * 6
END TYPE

' Deklariere Prozeduren, die aufgerufen werden:
DECLARE FUNCTION Filter$ (Anfrage$)
DECLARE FUNCTION FindeSatz% (TeilNummer$, SatzVar AS
BestPosten)

DECLARE SUB AddSatz (SatzVar AS BestPosten)
DECLARE SUB GibSatz (SatzVar AS BestPosten)
DECLARE SUB SchreibSatz (SatzVar AS BestPosten)
DECLARE SUB SortIndex ()
DECLARE SUB ZeigTeilNummer ()

' Definiere einen Puffer (der den Typ BestPosten
' verwendet) und definiere und dimensioniere das
' Indexdatenfeld:
DIM BestSatz AS BestPosten, Index(1 TO 100) AS IndexTyp

' Öffne die Direktzugriffsdatei:
OPEN "bestand.dat" FOR RANDOM AS #1 LEN = LEN(BestSatz)

' Berechne Anzahl der Datensätze in der Datei:
AnzSaetze = LOF(1) \ LEN(BestSatz)

' Wenn Datensätze vorhanden sind, lies sie und bilde den
' Index.

IF AnzSaetze <> 0 THEN
   FOR SatzNummer = 1 TO AnzSaetze
      ' Lies die Daten eines neuen Satzes der Datei:
      GET #1, SatzNummer, BestSatz
      ' Plaziere Teilenummer und Satznummer im Index:
      Index(SatzNummer).SatzNummer = SatzNummer
      Index(SatzNummer).TeilNummer = BestSatz.TeilNummer
   NEXT
   SortIndex        ' Sortiere Index in der Reihenfolge der
                    ' Teilenummern.
END IF
```

```
DO                       ' Schleife des Hauptmenüs
   CLS
   PRINT "(F)üge Datensätze hinzu."
   PRINT "(Z)eige Datensätze an."
   PRINT "(B)eende Programm."
   PRINT
   LOCATE , , 1
   PRINT "Wählen Sie (F, Z, oder B) hier: ";
   ' Warte, bis Benutzer F, Z oder B betätigt:
   DO
      Wahl$ = UCASE$(INPUT$(1))
   LOOP WHILE INSTR("FZB", Wahl$) = 0
   ' Verzweige entsprechend der Wahl:
   SELECT CASE Wahl$
      CASE "F"
         AddSatz BestSatz
      CASE "Z"
         IF AnzSaetze = 0 THEN
            PRINT : PRINT "Noch keine Datensätze in der
Datei.";
            PRINT " Weiter mit jeder Taste.";
            Pause$ = INPUT$(1)
         ELSE
            GibSatz BestSatz
         END IF
      CASE "B"          ' Beende Programm.
   END SELECT
LOOP UNTIL Wahl$ = "B"
CLOSE #1   ' Alles fertig; schließe Datei und beende.
END
'
' =========================ADDSATZ ==========================
'  Fügt an die Datei Datensätze aus Tastatureingabe an.
' ==========================================================
'
SUB AddSatz (SatzVar AS BestPosten) STATIC
   SHARED Index() AS IndexTyp, AnzSaetze
   DO
      CLS
      INPUT "Teilenummer : ", SatzVar.TeilNummer
      INPUT "Beschreibung: ", SatzVar.Beschreib
```

```
      ' Aufruf der FUNCTION Filter$, um Preis & Menge
      ' einzugeben:
      SatzVar.EinzPreis = VAL(Filter$("Einzelpreis : "))
      SatzVar.Menge = VAL(Filter$("Menge      : "))

      AnzSaetze = AnzSaetze + 1

      PUT #1, AnzSaetze, SatzVar

      Index(AnzSaetze).SatzNummer = AnzSaetze
      Index(AnzSaetze).TeilNummer = SatzVar.TeilNummer
      PRINT : PRINT "Weiter anfügen? ";
      OK$ = UCASE$(INPUT$(1))
   LOOP WHILE OK$ = "J"

   SortIndex            ' Sortiere Index-Datei erneut.
END SUB
'
' ========================= FILTER ==========================
'       Filtert alle nicht-numerischen Zeichen aus
'       einer Zeichenkette und gibt die gefilterte
'       Zeichenkette zurück.
' ===========================================================
'
FUNCTION Filter$ (Anfrage$) STATIC
   WertTemp2$ = ""
   PRINT Anfrage$;    ' Gib die übergebene Anfrage aus.
   INPUT "", WertTemp1$    ' Gib eine Zahl als
                           ' Zeichenkette an.
   KettLaenge = LEN(WertTemp1$)  ' Ermittle die Länge
                                 ' der Zeichenkette.
   FOR I% = 1 TO KettLaenge      ' Gehe Zeichen für
                                 ' Zeichen durch die
                                 ' Zeichenkette.

      Zeich$ = MID$(WertTemp1$, I%, 1)
      ' Ist das Zeichen ein gültiger Teil einer Zahl
      ' (d. h., eine Ziffer oder ein Dezimalpunkt)?
      ' Wenn ja, füge es an das Ende einer neuen
      ' Zeichenkette:
      IF INSTR("0.0123456789", Zeich$) > 0 THEN
         WertTemp2$ = WertTemp2$ + Zeich$

      ' Andernfalls prüfe, ob es ein kleines "l" ist,
      ' da an Schreibmaschinen gewohnte Benutzer eine
      ' 1 vielleicht auf diese Art eingeben:
      ELSEIF Zeich$ = "l" THEN
         WertTemp2$ = WertTemp2$ + "1" 'Ändere das "l"
                                       'in eine "1".
```

```
      END IF
   NEXT I%
   Filter$ = WertTemp2$          ' Gib die gefilterte
                                 ' Zeichenkette zurück.

END FUNCTION
'
' ====================== FINDESATZ ==========================
'     Verwendet einen binären Suchalgorithmus, um
'     einen Satz im Index ausfindig zu machen.
' ==========================================================
'
FUNCTION FindeSatz% (Teil$, SatzVar AS BestPosten) STATIC
   SHARED Index() AS IndexTyp, AnzSaetze

   ' Setze Ober- und Untergrenze für die Suche:
   ObSatz = AnzSaetze
   UntSatz = 1

   ' Suche, bis oberer Bereich kleiner als unterster
   ' ist:
   DO UNTIL (ObSatz< UntSatz)
      ' Wähle Mittelpunkt:
      Mittpunkt = (ObSatz + UntSatz) \ 2

      ' Überprüfe, ob dies der gewünschte Satz ist
      ' (RTRIM$() entfernt nachfolgende Leerzeichen
      ' aus einer Zeichenkette fester Länge):
      Test$ = RTRIM$(Index(Mittpunkt).TeilNummer)

      ' Wenn dies so ist, verlasse die Schleife:
      IF Test$ = Teil$ THEN
         EXIT DO

      ' Andernfalls, wenn das, wonach gesucht wird,
      ' größer ist, bewege unteres Ende nach oben:
      ELSEIF Teil$ > Test$ THEN
        UntSatz = Mittpunkt + 1

      ' Andernfalls bewege oberes Ende nach unten:
      ELSE
         ObSatz = Mittpunkt - 1
      END IF
   LOOP
```

```
   ' Wenn Teil gefunden wurde, lies Satz aus der Datei
   ' unter Verwendung des Zeigers im Index ein und
   ' setze FindeSatz% auf WAHR:
  IF Test$ = Teil$ THEN
      GET #1, Index(Mittpunkt).SatzNummer, SatzVar
      FindeSatz% = WAHR

   ' Andernfalls, wenn Teil nicht gefunden wurde,
   ' setze FindeSatz% auf FALSCH:
   ELSE
      FindeSatz% = FALSCH
   END IF
END FUNCTION
'
' ========================== GIBSATZ ======================
'    GIBSATZ ruft zunächst ZEIGTEILNUMMER auf, das
'    ein Menü von Teilenummern im oberen Bereich des
'    Bildschirms ausgibt. Danach fordert GIBSATZ den
'    Benutzer auf, eine Teilenummer einzugeben.
'    Abschließend ruft es die Prozeduren FINDESATZ und
'    SCHREIBSATZ auf, um den gegebenen Satz zu finden
'    und auszugeben.
' =========================================================
'
SUB GibSatz (SatzVar AS BestPosten) STATIC
   CLS
   ZeigTeilNummer   ' Rufe die SUB ZeigTeilNummer auf.

   ' Gib die Daten des angegebenen Satzes im
   ' unteren Teil des Bildschirms aus:
   DO
      PRINT "Geben Sie eine der oben aufgelisteten";
      PRINT " Teilenummern an (oder B zum Beenden) und "
      INPUT "betätigen Sie die <EINGABETASTE>: ", Teil$

      IF UCASE$(Teil$) <> "Q" THEN
         IF FindeSatz(Teil$, SatzVar) THEN
             SchreibSatz SatzVar
         ELSE
             PRINT "Teil nicht gefunden."
         END IF
      END IF
      PRINT STRING$(40, "_")
   LOOP WHILE UCASE$(Teil$) <> "B"

   VIEW PRINT  ' Stelle gesamten Bildschirm als Text-
               ' Darstellungsfeld wieder her.
```

```
END SUB
'
' ===================== SCHREIBSATZ =====================
'       Gibt einen Satz auf den Bildschirm aus
' ======================================================
'
SUB SchreibSatz (SatzVar AS BestPosten) STATIC
   PRINT "Teilenummer : "; SatzVar.TeilNummer
   PRINT "Beschreibung: "; SatzVar.Beschreib
   PRINT USING "Einzelpreis:$$###.##"; SatzVar.EinzPreis
   PRINT "Menge   :"; SatzVar.Menge
END SUB
'
' ==================== ZEIGTEILNUMMER ===================
'  Gibt einen Index aller Teilenummern im oberen
'  Teil des Bildschirms aus.
' ======================================================
'
SUB ZeigTeilNummer STATIC
   SHARED Index() AS IndexTyp, AnzSaetze
   CONST ANZSPALTEN = 8, SPALTENBREIT = 80 \ ANZSPALTEN
   ' Gib im oberen Teil des Bildschirms ein Menü aus,
   ' das alle Teilenummern für Datensätze in der Datei
   ' indiziert. Dieses Menü wird in Spalten gleicher
   ' Länge ausgegeben (Ausnahme ist vielleicht die
   ' letzte Spalte, die kürzer als die anderen sein
   ' kann):
   SpalteLang = AnzSaetze
   DO WHILE SpalteLang MOD ANZSPALTEN
      SpalteLang = SpalteLang + 1
   LOOP
   SpalteLang = SpalteLang \ ANZSPALTEN
   Spalte = 1
   SatzNummer = 1
   DO UNTIL SatzNummer > AnzSaetze
      FOR Zeile = 1 TO SpalteLang
         LOCATE Zeile, Spalte
         PRINT Index(SatzNummer).TeilNummer
         SatzNummer = SatzNummer + 1
         IF SatzNummer > AnzSaetze THEN EXIT FOR
      NEXT Zeile
      Spalte = Spalte + SPALTENBREIT
```

```
    LOOP
    LOCATE SpalteLang + 1, 1
    PRINT STRING$(80, "_")    ' Gib Trennzeile aus.
    ' Rolle Information über die Datensätze unter dem
    ' Teilenummern-Menü (auf diese Weise werden die
    ' Teilenummern nicht gelöscht):
    VIEW PRINT SpalteLang + 2 TO 24
END SUB
'
' ===================== SORTINDEX =======================
'              Sortiert den Index nach Teilenummern
' =========================================================
'
SUB SortIndex STATIC
    SHARED Index() AS IndexTyp, AnzSaetze
    ' Setze Vergleichsoffset gleich der Hälfte
    ' der Anzahl der Datensätze im Index
    Offset = AnzSaetze \ 2
    ' Wiederhole, bis Offset null wird:
    DO WHILE Offset > 0
       Limit = AnzSaetze - Offset
       DO
           ' Nimm an, kein Wechsel an diesem Offset:
           Wechsel = FALSCH
           ' Vergleiche Elemente und wechsele, wenn
           ' eines außerhalb der Reihenfolge:
           FOR I = 1 TO Limit
               IF Index(I).TeilNummer > Index(I +
Offset).TeilNummer THEN
                  SWAP Index(I), Index(I + Offset)
                  Wechsel = I
               END IF
           NEXT I
           ' Sortiere im nächsten Schritt nur bis dahin,
           ' wo letzter Wechsel vorgenommen wurde:
           Limit = Wechsel
       LOOP WHILE Wechsel
       ' Keine Wechsel beim letzten Offset, versuche es
       ' mit einem halb so großen:
       Offset = Offset \ 2
    LOOP
END SUB
```

Ausgabe

```
max144    max560    pat300    pat550    rw3300    rw5500    sim101    sim204
max321    max909    pat440    pat761    rw3301    rw5501    sim202    sim789
------------------------------------------------------------------------------
Geben Sie eine der oben aufgelisteten Teilenummern an (oder B zum Beenden)
und betätigen Sie die <EINGABETASTE>: sim204
Teilenummer : sim204
Beschreibung: Tretroller
Einzelpreis:  $80.00
Menge   : 5
-------------------------------------
Geben Sie eine der oben aufgelisteten Teilenummern an (oder B zum Beenden)
und betätigen Sie die <EINGABETASTE>: pat551
Teil nicht gefunden.
-------------------------------------
Geben Sie eine der oben aufgelisteten Teilenummern an (oder B zum Beenden)
und betätigen Sie die <EINGABETASTE>: pat550
Teilenummer : pat550
Beschreibung: t
Einzelpreis:    $0.00
Menge   : 0
-------------------------------------
Geben Sie eine der oben aufgelisteten Teilenummern an (oder B zum Beenden)
und betätigen Sie die <EINGABETASTE>: b
```

3.6.3 Terminal-Emulator (*terminal.bas*)

Das folgende einfache Programm wandelt den Computer in ein "dummes" Terminal um; das bedeutet, daß es den Computer in Zusammenarbeit mit einem Modem nur als E/A-Gerät funktionieren läßt. Dieses Programm verwendet die Anweisung **OPEN "COM1:"** und entsprechende Geräte-E/A-Funktionen zur Durchführung nachstehender Aufgaben:

1. Senden der eingegebenen Zeichen an das Modem.
2. Ausgabe der von dem Modem zurückgegebenen Zeichen auf den Bildschirm.

Es ist zu beachten, daß eingegebene Zeichen, die auf dem Bildschirm erscheinen, zuerst an das Modem gesendet und anschließend über den geöffneten Datenübertragungskanal an den Computer zurückgegeben werden. Dies kann über ein Modem mit Receive Detect (RD)- und Send Detect (SD)-Lämpchen überprüft werden, da diese in der gleichen Folge aufleuchten, wie Sie eintippen.

Zum Wählen einer Nummer müssen die Befehle Attention Dial Touch-Tone (ATDT, für Tastentelefon) oder Attention Dial Pulse (ADTP, für Wählscheibentelefon) über die Tastatur eingegeben werden (unter der Voraussetzung, daß es sich um ein Hayes-kompatibles Modem handelt).

Alle anderen zu dem Modem gesandten Befehle müssen ebenfalls über die Tastatur eingegeben werden.

Verwendete Anweisungen und Funktionen

Dieses Programm veranschaulicht folgende zur Datenübertragung mit einem Modem über den seriellen Anschluß des Computers benutzte Funktionen und Anweisungen:

- **OPEN "COM1:"**
- **EOF**
- **INPUT$**
- **LOC**

Programm-Listing

```
DEFINT A-Z

DECLARE SUB Filter (InZeichKett$)

COLOR 7, 1  ' Setze Bildschirmfarbe.
CLS

Ende$ = CHR$(0) + CHR$(16) ' Von INKEY$ angegebener
                           ' Wert, wenn Alt+q betätigt
                           ' wird.

' Gib Anfrage in unterster Bildschirmzeile aus und
' schalte den Cursor ein:
LOCATE 24, 1, 1
PRINT STRING$(80, "_");
LOCATE 25, 1
PRINT TAB(30); "Drücke Alt+q zum Beenden";

VIEW PRINT 1 TO 23    ' Ausgaben zwischen den Zeilen 1
                      ' & 23.

' Eröffne Kommunikation (1200 Baud, keine Parität,
' 8-Bit für Daten, 1 Stopbit, 256-Byte
' Eingabepuffer):
OPEN "COM1:1200,N,8,1" FOR RANDOM AS #1 LEN = 256

DO            ' Hauptschleife der Datenübertragung.

   TastEingab$ = INKEY$  ' Überprüfe Tastatur.

   IF TastEingab$ = Ende$ THEN   ' Verlasse die
                                 ' Schleife, wenn der
                                 ' Benutzer Alt+q
      EXIT DO                    ' betätigt hat.
```

```
   ELSEIF TastEingabs$ <> "" THEN ' Andernfalls, wenn
                                 ' der Benutzer eine
                                 ' Taste betätigt hat,
                                 ' sende das einge-
      PRINT #1, TastEingab$;     ' tippte Zeichen zum
   END IF                        ' Modem.

   ' Überprüfe das Modem. Wenn Zeichen warten (EOF(1)
   ' ist wahr), hole diese und gib sie auf den
   ' Bildschirm aus:
   IF NOT EOF(1) THEN

     'LOC(1) gibt die Anzahl der wartenden Zeichen an:
      ModemEingab$ = INPUT$(LOC(1), #1)

      Filter ModemEingab$ ' Filtere Zeilenvorschübe
                          ' und Rückschritte aus,
      PRINT ModemEingab$; ' schreibe dann.
   END IF
LOOP

CLOSE           ' Ende der Datenübertragung.
CLS
END
'
' ========================== FILTER ========================
'      Filtert Zeichen einer eingegebenen Zeichenkette
' ==========================================================
'
SUB Filter (InZeichKett$) STATIC

   ' Suche nach Rückschrittzeichen (CHR$(8)) und
   ' kodiere diese erneut in CHR$(29) um (Cursor-Taste
   ' nach links):
   DO
      RueckSchr = INSTR(InZeichKett$, CHR$(8))
      IF RueckSchr THEN
          MID$(InZeichKett$, RueckSchr) = CHR$(29)
      END IF
   LOOP WHILE RueckSchr

   ' Suche nach Zeilenvorschubzeichen (CHR$(10)) und
   ' entferne alle gefundenen:
```

```
   DO
      ZeilVor = INSTR(InZeichKett$, CHR$(10))
      IF ZeilVor THEN
         InZeichKett$ = LEFT$(InZeichKett$, ZeilVor - 1)_
                      + MID$(InZeichKett$, ZeilVor + 1)
      END IF
   LOOP WHILE ZeilVor
END SUB
```

4 Zeichenkettenverarbeitung

Dieses Kapitel zeigt, wie als Zeichenketten bezeichnete Folgen von ASCII-Zeichen zu bearbeiten sind. Die Manipulation von Zeichenketten ist unverzichtbar beim Verarbeiten von Textdateien, Sortieren von Daten oder Modifizieren eingegebener Zeichenkettendaten.

Nach Abschluß dieses Kapitels, werden Sie folgende, mit der Verarbeitung von Zeichenketten verbundene Aufgaben durchführen können:

- Deklarieren von Zeichenketten fester Länge.
- Vergleichen von Zeichenketten und Sortieren in alphabetischer Reihenfolge.
- Suchen nach einer Zeichenkette in einer anderen.
- Separieren der Teile einer Zeichenkette.
- Abschneiden führender oder nachfolgender Leerzeichen einer Zeichenkette.
- Erzeugen einer Zeichenkette durch Wiederholung eines einzelnen Zeichens.
- Umwandeln der Großbuchstaben einer Zeichenkette in Kleinbuchstaben und umgekehrt.
- Umwandeln numerischer Ausdrücke in Zeichenkettendarstellung und umgekehrt.

4.1 Definition der Zeichenketten

Eine Zeichenkette ist eine zusammenhängende Folge von Zeichen. Beispiele für Zeichen sind die Buchstaben des Alphabets (a-z bzw. A-Z), Interpunktionszeichen wie Kommata (,) oder Fragezeichen (?) und andere Symbole aus den Gebieten Mathematik und Wirtschaft, wie Plus- (+) oder Prozentzeichen (%).

In diesem Kapitel kann sich der Ausdruck "Zeichenkette" auf einen der folgenden Punkte beziehen:

- Eine Zeichenkettenkonstante.
- Eine Zeichenkettenvariable.
- Ein Zeichenkettenausdruck.

Zeichenkettenkonstanten werden auf eine der folgenden Arten deklariert:

1. Durch Einschließen einer Zeichenfolge zwischen doppelten Anführungszeichen, wie in nachstehender **PRINT**-Anweisung:

   ```
   PRINT "Ich verarbeite die Datei. Bitte warten."
   ```

   Dies wird als "literale Zeichenkettenkonstante" bezeichnet.

2. Durch die Gleichsetzung eines Namens mit einer literalen Zeichenkette in einer **CONST**-Anweisung, wie im folgenden gezeigt:

   ```
   'Definiere die Zeichenkettenkonstante MELDUNG:
   CONST MELDUNG = "Laufwerk nicht verriegelt."
   ```

   Dies wird als "symbolische Zeichenkettenkonstante" bezeichnet.

Zeichenkettenvariablen können auf eine der folgenden drei Arten deklariert werden:

1. Durch Anhängen des Zeichenkettensuffixes ($) an den Variablennamen:

   ```
   LINE INPUT "Geben Sie Ihren Namen ein: "; Puffer$
   ```

2. Durch Verwendung der Anweisung **DEFSTR**:

   ```
   ' Alle Variablen beginnend mit dem Buchstaben "P" sind
   ' Zeichenketten, es sei denn, sie enden mit einem der
   ' Suffixe für numerischen Typ (%, &, ! oder #):
   DEFSTR P
   .
   .
   .
   Puffer = "Ihr Name"    ' Puffer ist eine
                          ' Zeichenkettenvariable.
   ```

3. Durch Deklaration des Zeichenkettennamens in einer **AS STRING**-Anweisung:

   ```
   DIM Puffer1 AS STRING     ' Puffer ist eine Zeichenkette
                             ' variabler Länge.
   DIM Puffer2 AS STRING*10  ' Puffer2 ist eine 10-Byte lange
                             ' Zeichenkette fester Länge.
   ```

Ein Zeichenkettenausdruck ist eine Kombination von Zeichenkettenvariablen, -konstanten und/oder -funktionen.

Natürlich sind die Zeichen einer Zeichenkette im Speicher des Computers nicht in einer allgemein für Menschen erkennbaren Form gespeichert. Stattdessen wird jeder Buchstabe in eine Zahl übersetzt, die als ASCII-Code für dieses Zeichen bezeichnet wird. Der Großbuchstabe "A" ist beispielsweise als dezimal 65 (oder hexadezimal 41H) gespeichert, während der Kleinbuchstabe "a" dezimal 97 (oder hexadezimal 61H) entspricht. Eine komplette Liste der ASCII-Codes zu den entsprechenden Zeichen finden Sie in Anhang D, "Tastaturabfrage- und ASCII-Zeichencodes".

Der ASCII-Code eines Zeichens kann auch anhand der BASIC-Funktion **ASC** festgestellt werden; zum Beispiel gibt `ASC ("A")` den Wert 65 zurück. Die Umkehrung der **ASC**-Funktion ist die **CHR$**-Funktion. **CHR$** hat einen ASCII-Code als Argument und gibt das Zeichen dieses Codes aus; zum Beispiel zeigt die Anweisung `PRINT CHR$ (64)` das Zeichen @ an.

4.2 Zeichenketten variabler und fester Länge

In früheren BASIC-Versionen hatten Zeichenketten immer variable Längen. BASIC unterstützt jetzt sowohl Zeichenketten mit variabler als auch mit fester Länge.

4.2.1 Zeichenketten variabler Länge

Zeichenketten variabler Länge sind "elastisch"; das heißt, sie verkürzen und verlängern sich, um unterschiedliche Anzahlen von Zeichen zu speichern (von 0 bis höchstens 32.767 Zeichen). Die Länge der Variablen `Temp$` im folgenden Beispiel verändert sich je nach der Größe dessen, was in `Temp$` gespeichert ist:

```
Temp$ = "1234567"

' Die Funktion LEN gibt die Länge einer Zeichenkette (Anzahl
' von Zeichen, die diese enthält) an:
PRINT LEN(Temp$)

Temp$ = "1234"
PRINT LEN(Temp$)
```

Ausgabe

```
 7
 4
```

4.2.2 Zeichenketten fester Länge

Zeichenketten fester Länge werden normalerweise als Datensatzelemente in einem mit **TYPE...END TYPE** benutzerdefinierten Datentyp verwendet. Dennoch können sie auch selbst in **COMMON**-, **DIM**-, **REDIM**-, **SHARED**- oder **STATIC**-Anweisungen wie in der folgenden Anweisung deklariert sein:

```
DIM Puffer AS STRING * 10
```

Zeichenketten fester Länge haben immer, wie der Name schon sagt, eine konstante Länge, unabhängig von der darin gespeicherten Zeichenkettenlänge. Dies ist aus der Ausgabe des folgenden Beispiels ersichtlich:

```
DIM NachName AS STRING * 8
DIM VorName AS STRING * 10

NachName = "Heiligensetzer"
VorName  = "Stefan"

PRINT "123456789012345678901234567890"
PRINT VorName; NachName
PRINT LEN(VorName)
PRINT LEN(NachName)
```

Ausgabe

```
123456789012345678901234567890
Stefan    Heiligen
 10
 8
```

Beachten Sie, daß sich die Längen der Zeichenkettenvariablen `VorName` und `NachName` nicht geändert haben, was durch die von der Funktion **LEN** angegebenen Werte (und die vier Leerzeichen nach dem Namen mit sechs Buchstaben, Stefan) gezeigt wird.

Die Ausgabe des obigen Beispiels verdeutlicht ebenfalls, wie Werte, die Variablen fester Länge zugewiesen werden, linksbündig angeordnet und, falls erforderlich, rechts gekürzt werden. Es ist nicht nötig, die **LSET**-Funktion zur linksbündigen Anordnung von Werten in Zeichenketten fester Länge zu verwenden, da dieser Vorgang automatisch durchgeführt wird. Wenn Sie eine Zeichenkette innerhalb einer Zeichenkette fester Länge rechtsbündig anordnen möchten, ist **RSET** wie folgt zu benutzen:

```
DIM NamePuffer AS STRING * 10
RSET NamePuffer = "Stefan"
PRINT "1234567890"
PRINT NamePuffer
```

Ausgabe

```
1234567890
    Stefan
```

4.3 Kombinieren von Zeichenketten

Zwei Zeichenketten können mit Hilfe des Operators Plus (+) zusammengesetzt werden. Die dem Pluszeichen folgende Zeichenkette wird der dem Pluszeichen vorhergehenden Zeichenkette hinzugefügt, wie im nächsten Beispiel gezeigt:

```
A$ = "erste Zeichenkette"
B$ = "zweite Zeichenkette"
C$ = A$ + B$
PRINT C$
```

Ausgabe

```
erste Zeichenkettezweite Zeichenkette
```

Das Zusammensetzen von Zeichenketten auf diese Art wird als "Verketten" (Konkatenation) bezeichnet.

Es ist zu beachten, daß im vorhergehenden Beispiel beide Zeichenketten ohne dazwischenliegende Leerzeichen zusammengesetzt sind. Wenn Sie ein Leerzeichen einsetzen möchten, kann eine der Zeichenketten `A$` oder `B$` wie folgt ausgefüllt werden:

```
B$ = " zweite Zeichenkette"     'Führendes Leerzeichen in B$
```

Da Werte in Zeichenketten fester Länge linksbündig angeordnet sind, kann es vorkommen, daß Sie beim Zusammensetzen der Zeichenketten zusätzliche Leerzeichen wie im nächsten Beispiel finden:

```
TYPE NameTyp
   Vor  AS STRING * 15
   Nach AS STRING * 12
END TYPE

DIM NameSatz AS NameTyp

' Die Konstante "Christiane" ist in der Variablen
' NameSatz.Vor linksbündig angeordnet, so daß 5 nachfolgende
' Leerzeichen existieren:
NameSatz.Vor  = "Christiane"
NameSatz.Nach = "Kling"

' Gib eine Zeile mit Zahlen als Bezug aus:
PRINT "123456789012345678901234567890"

GesamtName$ = NameSatz.Vor + NameSatz.Nach
PRINT GesamtName$
```

Ausgabe

```
123456789012345678901234567890
Christiane     Kling
```

Die Funktion **LTRIM$** schneidet führende Leerzeichen von einer Zeichenkette ab, während die **RTRIM$**-Funktion nachfolgende Leerzeichen abschneidet, wobei die Originalzeichenkette unverändert erhalten bleibt. (Weitere Informationen zu diesen Funktionen finden Sie in Abschnitt 4.6,"Ausgrenzen von Zeichenkettenteilen".)

4.4 Vergleichen von Zeichenketten

Zeichenketten können anhand folgender Vergleichsoperatoren verglichen werden:

Operator	***Bedeutung***
<>	ungleich
=	gleich
<	kleiner
>	größer
<=	kleiner gleich
>=	größer gleich

Ein einzelnes Zeichen ist größer als ein anderes, wenn sein ASCII-Wert größer ist. Zum Beispiel ist der ASCII-Wert des Buchstabens "B" größer als der Wert des Buchstabens "A", so daß der Ausdruck `"B" > "A"` wahr ist.

Beim Vergleich zweier Zeichenketten prüft BASIC die ASCII-Werte sich entsprechender Zeichen. Das erste Zeichen, durch das sich die beiden Zeichenketten unterscheiden, bestimmt die alphabetische Reihenfolge der Zeichenketten. Zum Beispiel sind die Zeichenketten `"Verdichter"` bzw. `"Verdichtung"` bis zum neunten Zeichen identisch (`"e"` bzw. `"u"`). Da der ASCII-Wert von `"e"` kleiner als der ASCII-Wert von `"u"` ist, ist der Ausdruck `"Verdichter" < "Verdichtung"` wahr. Beachten Sie, daß die ASCII-Werte für Buchstaben der alphabetischen Reihenfolge von A bis Z bzw. a bis z folgen, so daß die oben aufgeführten Vergleichsoperatoren zum Alphabetisieren von Zeichenketten verwendet werden können. Darüber hinaus haben Großbuchstaben kleinere ASCII-Werte als Kleinbuchstaben, so daß in einer sortierten Liste `"ASCII"` vor `"ascii"` stehen würde.

Wenn es keinen Unterschied zwischen sich entsprechenden Zeichen zweier Zeichenketten gibt und beide die gleiche Länge haben, sind die beiden Zeichenketten gleich. Wenn es keinen Unterschied zwischen sich entsprechenden Zeichen zweier Zeichenketten gibt, eine der Zeichenketten jedoch länger ist, dann ist die längere Zeichenkette größer als die kürzere. Zum Beispiel sind die Ausdrücke `"abc" = "abc"` und `"aaaaaaa" > "aaa"` beide wahr.

Bei dem Vergleich von Zeichenketten sind führende und nachfolgende Leerzeichen von Bedeutung. Die Zeichenkette `"   Test"` ist zum Beispiel kleiner als die Zeichenkette `"Test"`, da ein Leerzeichen (`" "`) kleiner als ein `"T"` ist; andererseits ist die Zeichenkette `"Test   "` größer als die Zeichenkette `"Test"`. Diese Tatsachen sind beim Vergleich von Zeichenketten fester und variabler Länge zu berücksichtigen, da die Zeichenketten fester Länge nachfolgende Leerzeichen enthalten können.

4.5 Suchen nach Zeichenketten

Eine der häufigsten Aufgaben der Zeichenkettenverarbeitung ist die Suche nach einer Zeichenkette innerhalb einer anderen Zeichenkette. Die Funktion **INSTR** (*Zeichenkette1*, *Zeichenkette2*) gibt an, ob *Zeichenkette2* in *Zeichenkette1* enthalten ist oder nicht, indem sie die Position des ersten Zeichens in *Zeichenkette1* zurückgibt, ab der beide übereinstimmen. Siehe nächstes Beispiel:

```
ZeichKett1$ = "Eine Textzeile, die 39 Zeichen enthält."
ZeichKett2$ = "Zeichen"

PRINT "         1         2         3         4"
PRINT "1234567890123456789012345678901234567890"
PRINT ZeichKett1$
PRINT ZeichKett2$
PRINT INSTR(ZeichKett1$, ZeichKett2$)
```

Ausgabe

```
         1         2         3         4
1234567890123456789012345678901234567890
Eine Textzeile, die 39 Zeichen enthält.
Zeichen
 24
```

Wenn keine Übereinstimmung gefunden wird (das heißt, *Zeichenkette2* gehört nicht zu *Zeichenkette1*), gibt **INSTR** den Wert 0 zurück.

Eine Abwandlung der oben gezeigten Syntax, **INSTR** (*Position, Zeichenkette1, Zeichenkette2*), erlaubt die Angabe der Position, ab der die Suche in *Zeichenkette1* beginnen soll. Zur Verdeutlichung verändert die folgende Abwandlung des vorhergehenden Beispiels die Ausgabe:

```
' Beginne die Suche nach einer Übereinstimmung mit dem
' 30-ten Zeichen von ZeichKett1$:
PRINT INSTR(30, ZeichKett1$, ZeichKett2$)
```

Ausgabe

```
         1         2         3         4
1234567890123456789012345678901234567890
Eine Textzeile, die 39 Zeichen enthält.
Zeichen
 0
```

Mit anderen Worten, erscheint die Zeichenkette `"Zeichen"` nicht nach dem 30-ten Zeichen von `ZeichKett1$`.

Die Abwandlung **INSTR** (*Position, Zeichenkette1, Zeichenkette2*) ermöglicht es, nicht nur das erste, sondern jedes Vorkommen der *Zeichenkette2* in *Zeichenkette1* zu finden, wie im nächsten Beispiel gezeigt:

```
ZeichKett1$ = "der Gerd, der taucht nicht."
ZeichKett2$ = "der"
PRINT ZeichKett1$
Start       = 1
AnzUeberein = 0
DO
   Ueberein = INSTR(Start, ZeichKett1$, ZeichKett2$)
   IF Ueberein > 0 THEN
      PRINT TAB(Ueberein); ZeichKett2$
      Start       = Ueberein + 1
      AnzUeberein = AnzUeberein + 1
   END IF
LOOP WHILE Ueberein
PRINT "Anzahl der Übereinstimmungen ="; AnzUeberein
```

Ausgabe

```
der Gerd, der taucht nicht.
der
          der
Anzahl der Übereinstimmungen = 2
```

4.6 Ausgrenzen von Zeichenkettenteilen

Abschnitt 4.3., "Kombinieren von Zeichenketten", zeigt, wie Zeichenketten mit dem Pluszeichen zusammenzusetzen (zu verketten) sind. In BASIC sind mehrere Funktionen verfügbar, die Zeichenketten auftrennen bzw. Teile einer Zeichenkette entweder von der linken oder von der rechten Seite, oder aus der Mitte einer Zielzeichenkette zurückgeben.

4.6.1 Ausgrenzen der Zeichen von der linken Seite einer Zeichenkette

Die Funktionen **LEFT$** und **RTRIM$** geben Zeichen von der linken Seite einer Zeichenkette zurück. Die Funktion **LEFT$** (*Zeichenkette, Anzahl*) gibt eine gegebene *Anzahl* von Zeichen von der linken Seite der *Zeichenkette* aus. Zum Beispiel:

```
Test$ = "Aktenordner"
' Schreibe die 5 äußerst linken Zeichen von Test$:
PRINT LEFT$(Test$, 5)
```

Ausgabe

```
Akten
```

Wie Sie feststellen können, verändert **LEFT$**, wie alle anderen in diesem Kapitel beschriebenen Funktionen, im vorhergehenden Beispiel nicht die ursprüngliche Zeichenkette `Test$`; es gibt nur eine andere Zeichenkette aus, die eine Kopie eines Teiles von `Test$` darstellt.

Die Funktion **RTRIM$** gibt den linken Teil einer Zeichenkette nach Entfernen aller Leerzeichen an ihrem Ende an. Vergleichen Sie zum Beispiel die Ausgaben der nächsten beiden **PRINT**-Anweisungen:

```
PRINT "eine linksbündige Zeichenkette  "; "nächste"
PRINT RTRIM$ ("eine linksbündige Zeichenkette  "); "nächste"
```

Ausgabe

```
eine linksbündige Zeichenkette  nächste
eine linksbündige Zeichenkettenächste
```

RTRIM$ ist beim Vergleich von Zeichenketten fester und variabler Länge, wie im nächsten Beispiel, behilfich:

```
DIM NameSatz AS STRING * 10
NameSatz = "Fred"
NameTest$ = "Fred"
```

```
' Verwende RTRIM$, um alle Leerzeichen von der rechten Seite
' der Zeichenkette fester Länge NameSatz zu entfernen,
' vergleiche sie dann mit der Zeichenkette variabler Länge
' NameTest$:
IF RTRIM$(NameSatz) = NameTest$ THEN
   PRINT "Sie sind gleich"
ELSE
   PRINT "Sie sind nicht gleich"
END IF
```

Ausgabe

```
Sie sind gleich
```

4.6.2 Ausgrenzen der Zeichen von der rechten Seite einer Zeichenkette

Die Funktionen **RIGHT$** und **LTRIM$** geben Zeichen von der rechten Seite einer Zeichenkette an. Die Funktion **RIGHT$** (*Zeichenkette, Anzahl*) liefert eine gegebene *Anzahl* Zeichen der rechten Seite von *Zeichenkette*. Zum Beispiel:

```
Test$ = "1234567890"
'Schreibe die 5 äußerst rechten Zeichen von Test$:
PRINT RIGHT$ (Test$,5)
```

Ausgabe

```
67890
```

Die Funktion **LTRIM$** gibt den rechten Teil einer Zeichenkette, nach Entfernen aller Leerzeichen an ihrem Ende an. Vergleichen Sie zum Beispiel die Ausgaben der nächsten zwei **PRINT**-Anweisungen:

```
ZeichKett1$ = "erst"
ZeichKett2$ = "      eine rechtsbündige ZeichKette"
PRINT ZeichKett1$; ZeichKett2$
PRINT ZeichKett1$;LTRIM (ZeichKett2$)
```

Ausgabe

```
erst      eine rechtsbündige Zeichenkette
ersteine rechtsbündige Zeichenkette
```

4.6.3 Ausgrenzen der Zeichen von einer beliebigen Stelle in einer Zeichenkette

Verwenden Sie die Funktion **MID$**, um eine beliebige Anzahl von Zeichen ab einem beliebigen Punkt aus einer Zeichenkette zurückzuholen. Die Funktion **MID$** (*Zeichenkette, Beginn, Anzahl*) gibt ab dem Zeichen mit der Position *Beginn* die *Anzahl* von Zeichen aus *Zeichenkette* an. Zum Beispiel beginnt die Anweisung

```
MID$("**über den Wolken**", 12, 6)
```

mit dem zwölften Zeichen (`W`) der Zeichenkette

```
**über den Wolken**
```

und liefert dieses Zeichen plus der nächsten fünf Zeichen (`Wolken`).

Das nächste Beispiel zeigt, wie die Funktion **MID$** zu verwenden ist, um eine Textzeile Zeichen für Zeichen zu durchgehen:

```
.
.
.
' Ermittle die Anzahl von Zeichen in der Textzeichenkette:
Laenge = LEN(TextZeichenKette$)
FOR I = 1 TO Laenge
  ' Nimm das erste Zeichen, dann das zweite, das dritte usw.
  ' bis zum Ende der Zeichenkette:
  Zeichen$ = MID$(TextZeichenKette$, I, 1)
  ' Werte dieses Zeichen aus:
  .
  .
  .
NEXT
```

4.7 Erzeugen von Zeichenketten

Die einfachste Möglichkeit, eine aus einem sich wiederholenden Buchstaben bestehende Zeichenkette zu erzeugen, ist die Verwendung der eingebauten Funktion **STRING$**. Die Funktion **STRING$** (*Anzahl, Zeichenkette*) erstellt eine neue Zeichenkette, die eine gegebene *Anzahl* von Zeichen lang ist und deren jeweilige Zeichen das erste Zeichen des *Zeichenkettenargumentes* bildet. Beispielsweise erzeugt die Anweisung

```
Fuellzeichen$ = STRING$(27, "*")
```

eine Zeichenkette von 27 Sternchen. Für Zeichen, die nicht durch Eintippen erzeugt werden können, wie die Zeichen, deren ASCII-Wert mehr als 127 beträgt, ist die alternative Form dieser Funktion, **STRING$**(*Anzahl,Code*), zu benutzen. Diese Form erzeugt eine Zeichenkette, die eine gegebene *Anzahl* von Zeichen lang ist, wobei jedes Zeichen, wie im nächsten Beispiel, den vom *Codeargument* angegebenen ASCII-Wert *Code* hat:

```
' Gib eine Zeichenkette von 10 "@" Zeichen aus (64 ist der
ASCII-Code für @):
PRINT STRING$(10, 64)
```

Ausgabe

```
@@@@@@@@@@
```

Die Funktion **SPACE$**(*Anzahl*) erzeugt eine Zeichenkette, die aus einer gegebenen *Anzahl* von Leerzeichen besteht.

4.8 Verändern der Groß- bzw. Kleinschreibung

Es kann vorkommen, daß Sie Großbuchstaben in Kleinbuchstaben umwandeln möchten und umgekehrt. Beispielsweise wäre diese Umsetzung zweckmäßig bei der Suche nach bestimmten Zeichenketten in einer großen Datei, in der es nicht auf Groß- bzw. Kleinschreibung ankommt (mit anderen Worten, `HILFE`, `hilfe` oder `Hilfe` würden alle als gleich betrachtet). Diese Funktion kann sich ebenfalls als nützlich erweisen, wenn Sie nicht sicher sind, ob ein Benutzer seine Eingabe in Groß- oder Kleinbuchstaben eintippt.

Die **UCASE$**- und **LCASE$**-Funktionen führen folgende Umwandlungen in einer Zeichenkette durch:

- **UCASE$** liefert eine Kopie der an sie übergebenen Zeichenkette, wobei alle Kleinbuchstaben in Großbuchstaben umgewandelt werden.
- **LCASE$** liefert eine Kopie der an sie übergebenen Zeichenkette, wobei alle Großbuchstaben in Kleinbuchstaben umgewandelt werden.

Beispiel

```
Muster$="* Der ASCII-Anhang: eine nützliche Code-Tabelle *"
PRINT Muster$
PRINT UCASE$(Muster$)
PRINT LCASE$(Muster$)
```

Ausgabe

```
* Der ASCII-Anhang: eine nützliche Code-Tabelle *
* DER ASCII-ANHANG: EINE NÜTZLICHE CODE-TABELLE *
* der ascii-anhang: eine nützliche code-tabelle *
```

Buchstaben, die bereits Großbuchstaben sind, Umlaute, wie auch Zeichen, die keine Buchstaben sind, werden von **UCASE$** nicht verändert; Kleinbuchstaben und Zeichen, die keine Buchstaben sind, sowie Umlaute und "ß" werden von der **LCASE$**-Funktion ebenfalls nicht verändert.

4.9 Zeichenketten und Zahlen

BASIC erlaubt weder, daß eine Zeichenkette einer numerischen Variablen, noch daß ein numerischer Ausdruck einer Zeichenkettenvariablen zugewiesen wird. Zum Beispiel führen beide der folgenden Anweisungen zu der Fehlermeldung `Unverträgliche Datentypen`:

```
TempPuffer$ = 45
Zaehler%    = "45"
```

Es ist stattdessen die Funktion **STR$** zur Angabe der Zeichenkettendarstellung einer Zahl, oder die Funktion **VAL** zur Angabe der numerischen Darstellung einer Zeichenkette zu benutzen.

```
' Die folgenden Anweisungen sind beide gültig:
TempPuffer$ = STR$(45)
Zaehler%    = VAL ("45")
```

Beachten Sie, daß **STR$** das führende Leerzeichen, das BASIC für positive Zahlen ausgibt, wie im nachstehenden kurzen Beispiel einfügt:

```
FOR I = 0 TO 9
   PRINT STR$(I);
NEXT
```

Ausgabe

```
 0 1 2 3 4 5 6 7 8 9
```

Dieses Leerzeichen kann anhand der Funktion **LTRIM$**, wie unten gezeigt, ausgelassen werden:

```
FOR I = 0 TO 9
   PRINT LTRIM$(STR$(I));
NEXT
```

Ausgabe

```
0123456789
```

Eine andere Möglichkeit zum Formatieren numerischer Ausgaben ist die Verwendung der Anweisung **PRINT USING**. (Weitere Informationen finden Sie in Abschnitt 3.1, "Ausgabe von Text auf den Bildschirm".)

4.10 Verändern von Zeichenketten

Die in den vorhergehenden Abschnitten erwähnten Funktionen lassen ihre Zeichenkettenargumente unverändert. Veränderungen werden nur in einer Kopie des Argumentes vorgenommen.

Im Gegensatz dazu verändert die **MID$**-Anweisung (nicht zu verwechseln mit der in Abschnitt 4.6.3, "Ausgrenzen der Zeichen von einer beliebigen Stelle in einer Zeichenkette", behandelten **MID$**-Funktion) ihr Argument durch Eingliedern einer anderen Zeichenkette in das Argument. Die eingegliederte Zeichenkette ersetzt einen Teil oder die gesamte ursprüngliche Zeichenkette, wie im folgenden Beispiel gezeigt:

```
Temp$ = "In der Sonne."
PRINT Temp$
' Ersetze das "I" durch ein "A":
MID$(Temp$,1) = "A"
' Ersetze "Sonne." durch "Straße.":
MID$(Temp$,8) = "Straße"
PRINT Temp$
```

Ausgabe

```
In der Sonne.
An der Straße.
```

4.11 Anwendungsbeispiel: Umsetzung einer Zeichenkette in eine Zahl (*zeiinzah.bas*)

Das folgende Programm liest eine als Zeichenkette eingegebene Zahl, filtert alle numerischen Zeichen (wie zum Beispiel Kommata) aus der Zeichenkette heraus und wandelt die Zeichenkette anschließend mit Hilfe der Funktion **VAL** in eine Zahl um.

Verwendete Anweisungen und Funktionen

Dieses Programm zeigt folgende in diesem Kapitel erläuterte Funktionen zur Verarbeitung von Zeichenketten:

- **INSTR**
- **LEN**
- **MID$**
- **VAL**

Programm-Listing

```
DECLARE FUNCTION Filter$ (Txt$, FilterZeichKett$)
' Gib eine Zeile ein:
LINE INPUT "Geben Sie eine Zahl mit Kommata ein: ", A$
```

```
' Suche nur nach gültigen numerischen Zeichen
' (0123456789.-) in der eingegebenen Zeichenkette:
GueltZahl$ = Filter$(A$, "0123456789.-")

' Wandle die Zeichenkette in eine Zahl um:
PRINT "Zahlenwert = " VAL(GueltZahl$)
END
'
' ==================== FILTER =====================
'      Entfernt aus einer Zeichenkette ungewollte
'      Zeichen, indem sie diese mit einer
'      Filterzeichenkette vergleicht, die nur
'      zulässige numerische Zeichen enthält
' ================================================
'
FUNCTION Filter$ (Txt$, FilterZeichKett$) STATIC
   Temp$ = ""
   TxtLaenge = LEN(Txt$)

   FOR I = 1 TO TxtLaenge      ' Isoliere jedes Zeichen
      C$ = MID$(Txt$, I, 1)    ' der Zeichenkette.

      ' Wenn sich das Zeichen in der
      ' Filterzeichenkette befindet, sichere es:
      IF INSTR(FilterZeichKett$, C$) <> 0 THEN
        Temp$ = Temp$ + C$
      END IF
   NEXT I

   Filter$ = Temp$
END FUNCTION
```

5 Graphiken

Dieses Kapitel zeigt, wie Graphikanweisungen und -funktionen von BASIC zu verwenden sind, um auf dem Bildschirm eine Vielzahl von Formen, Farben und Mustern zu erzeugen. Mit Graphik können Sie nahezu jedem Programm eine optische Dimension hinzufügen, ganz gleich, ob es sich um ein Spiel, ein Lernprogramm, eine wissenschaftliche Anwendung oder ein Finanzpaket handelt.

Nach Bearbeitung dieses Kapitels werden Sie in der Lage sein, folgende graphische Aufgaben durchzuführen:

- Verwenden des physikalischen Koordinatensystems Ihres Personal Computer-Bildschirms, um einzelne Bildpunkte zu positionieren, diese ein- oder auszuschalten und deren Farben zu verändern.
- Zeichnen von Geraden.
- Zeichnen und Ausfüllen einfacher Figuren wie Kreise, Ovale und Rechtecke.
- Begrenzen des die graphische Ausgabe anzeigenden Bildschirmbereiches durch Verwendung von Darstellungsfeldern.
- Justieren der zur Positionierung eines Bildpunktes verwendeten Koordinaten durch Neudefinierung von Bildschirmkoordinaten.
- Verwendung von Farbe bei graphischer Ausgabe.
- Erstellen von Mustern und deren Verwendung zum Ausfüllen umrandeter Abbildungen.
- Kopieren von Bildern und deren Reproduktion an einer anderen Bildschirmstelle.
- Bewegte graphische Ausgabe.

Es ist empfehlenswert den untenstehenden Abschnitt 5.1 durchzulesen, da er wichtige Informationen bezüglich der Graphikbeispiele in diesem Kapitel enthält.

5.1 Für graphische Programme erforderliche Voraussetzungen

Um die in diesem Kapitel gezeigten Beispiele zu starten, muß Ihr Computer entweder über eingebaute Graphikfähigkeit oder über eine graphische Adapterkarte verfügen, wie z. B. eine Farb-Graphikkarte (Color Graphics Adapter, CGA), einen Erweiterten Graphikadapter (Enhanced Graphics Adapter, EGA) oder einen Video Graphikadapter (VGA). Sie benötigen ebenfalls einen Bildschirm (entweder monochrom oder farbig), der aus Bildpunkten aufgebaute Graphiken unterstützt.

Berücksichtigen Sie bitte darüber hinaus, daß alle diese Programme einen die graphische Ausgabe unterstützenden "Bildschirmmodus" erfordern. (Der Bildschirmmodus steuert die Schärfe graphischer Bilder, die Anzahl der verfügbaren Farben und den Teil des anzuzeigenden Bildschirmspeichers.) Zur Auswahl eines graphischen Ausgabemodus ist folgende Anweisung vor jeglicher in diesem Kapitel aufgeführten Anweisung oder Funktion im Programm zu verwenden:

SCREEN *Modus*

Modus kann hier je nach der im Computer installierten Monitor/Adapterkombination 1, 2, 3, 4, 7, 8, 9, 10, 11, 12 oder 13 sein.

Vergewissern Sie sich durch folgende einfache Prüfung, daß die Benutzer Ihrer Programme über graphikunterstützende Hardware verfügen:

```
CONST FALSCH = 0, WAHR = NOT FALSCH
' Test zur Sicherstellung, daß der Benutzer Hardware mit
' Farb/Graphikfähigkeit hat:
ON ERROR GOTO Meld            ' Schalte Fehlerverfolgung ein.
SCREEN 1                      ' Versuche graphischen Modus
                              ' eins.
ON ERROR GOTO 0               ' Schalte Fehlerverfolgung aus.
IF KeineGraf THEN END         ' Beende, wenn keine graphische
                              ' Hardware.
.
.
.
END
' Fehlerbehandlungsroutine:
Meld:
   PRINT "Dieses Programm erfordert Graphikfähigkeit."
   KeineGraf = WAHR
   RESUME NEXT
```

Wenn der Benutzer lediglich einen monochromen Bildschirm ohne Graphikadapter hat, ergibt die Anweisung **SCREEN** einen Fehler, als dessen Folge eine Verzweigung in die Fehlerbehandlungsroutine `Meld` ausgelöst wird. (Weitere Informationen zur Fehlerverfolgung finden Sie in Kapitel 6, "Fehler- und Ereignisverfolgung".)

5.2 Bildpunkte und Bildschirmkoordinaten

Auf einem Bildschirm sind Figuren und Abbildungen aus einzelnen Lichtpunkten zusammengesetzt, die als Bildelemente oder als "Bildpunkte" (manchmal als "Pixel") bezeichnet werden. BASIC zeichnet und malt auf den Bildschirm, indem es diese Bildpunkte ein- oder ausschaltet sowie deren Farben verändert.

Ein typischer Bildschirm ist aus einem Raster von Bildpunkten zusammengesetzt. Die genaue Anzahl der Bildpunkte dieses Rasters hängt von der installierten Hardware und dem in der Anweisung **SCREEN** gewählten Bildschirmmodus ab. Je größer die Anzahl von einschaltbaren Bildpunkten, desto größer ist die Schärfe der graphischen Ausgabe. Eine Anweisung **SCREEN 1** ergibt z. B. eine Auflösung von 320 Bildpunkten horizontal mit 200 Punkten vertikal (320 x 200 Bildpunkte), während eine Anweisung **SCREEN 2** eine Auflösung von 640 x 200 Bildpunkten ergibt. Die höhere horizontale Dichte im Bildschirmmodus 2 - 640 Bildpunkte pro Zeile gegenüber 320 Bildpunkten pro Zeile - gibt Bildern gegenüber Bildschirmmodus 1 ein schärferes Erscheinungsbild.

Je nach der Graphikfähigkeit des Systems kann ein anderer Bildschirmmodus verwendet werden, der sogar höhere Auflösungen unterstützt (und weitere Bildschirmcharakteristika anpaßt). Nähere Informationen sind dem QB-Ratgeber zu entnehmen.

Wenn Ihr Bildschirm einen der erforderlichen graphischen Modi hat, kann jeder Bildpunkt mit Hilfe von Koordinatenpaaren positioniert werden. Die erste Zahl in jedem Koordinatenpaar gibt die Bildpunktnummer in der linken Seite des Bildschirms an, während die zweite Zahl in jedem Paar die Bildpunktnummer am oberen Bildschirmrand angibt. Im Bildschirmmodus 2 hat der Bildpunkt in der äußersten oberen linken Ecke des Bildschirms, zum Beispiel, die Koordinaten `(0, 0)`, der Punkt in Bildschirmmitte die Koordinaten `(320,100)` und der Punkt in der äußersten unteren rechten Ecke des Bildschirms die Koordinaten `(639,199)`, wie in Abbildung 5.1 gezeigt.

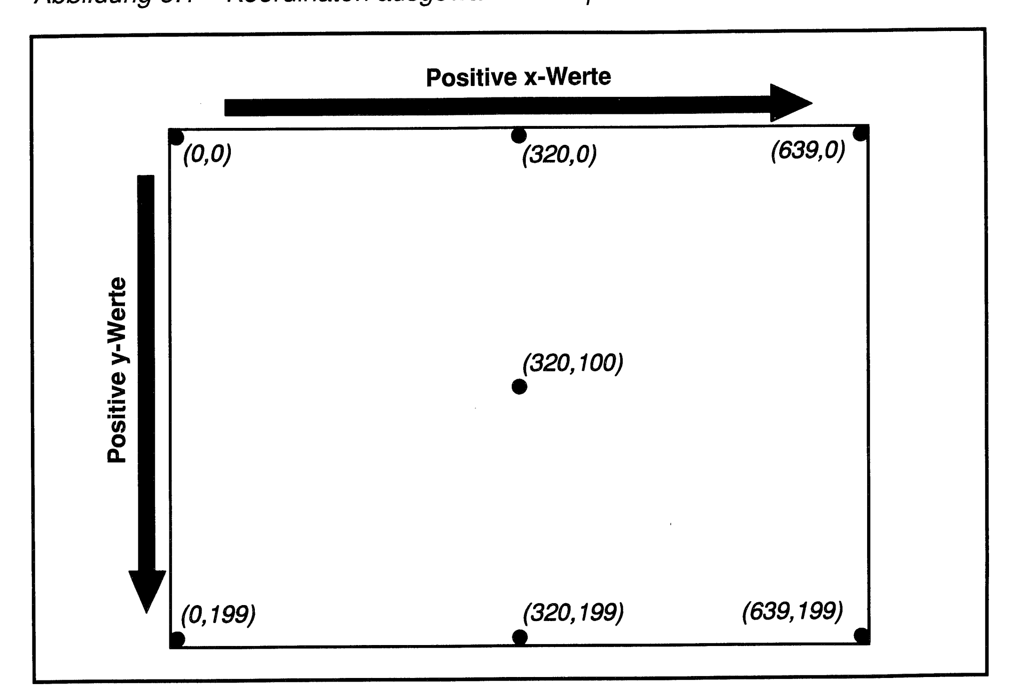
BASIC verwendet diese Bildschirmkoordinaten zur Bestimmung der Stelle, an der Graphiken angezeigt werden (zum Beispiel die Endpunkte einer Geraden oder der Mittelpunkt eines Kreises), wie in Abschnitt 5.3, "Das Zeichnen von Grundformen: Punkte, Geraden, Rechtecke und Kreise" gezeigt.

Graphische Koordinaten unterscheiden sich von Koordinaten im Textmodus, die in einer **LOCATE**-Anweisung angegeben werden. Zum einen ist **LOCATE** nicht so genau: Graphische Koordinaten zeigen auf einzelne Bildpunkte des Bildschirms, während von **LOCATE** verwendete Koordinaten Positionen von Zeichen sind. Zum anderen werden Koordinaten im Textmodus in der Form "Zeile, Spalte", wie im folgenden Beispiel angegeben:

```
' Spring in die 13-te Zeile, 15-te Spalte, gib dann die
' gezeigte Meldung aus:
LOCATE 13, 15
PRINT "Dies sollte in der Bildschirmmitte erscheinen."
```

Die graphischen Koordinaten werden, umgekehrt, in der Form "Spalte, Zeile" angegeben. Eine **LOCATE**-Anweisung hat keine Auswirkung auf die Positionierung graphischer Ausgaben auf dem Bildschirm.

Abbildung 5.1 Koordinaten ausgewählter Bildpunkte im Bildschirmmodus 2



5.3 Das Zeichnen von Grundformen: Punkte, Geraden, Rechtecke und Kreise

Sie können Koordinatenwerte an BASIC-Graphikanweisungen übergeben, um eine Vielzahl einfacher Figuren zu erstellen, wie in den Abschnitten 5.3.1 bis 5.3.2 gezeigt wird.

5.3.1 Graphische Darstellung von Punkten anhand von PSET und PRESET

Die unterste Ebene der Steuerung graphischer Ausgabe ist das einfache Ein- und Ausschalten einzelner Bildpunkte. In BASIC wird dieser Vorgang mit Hilfe der Anweisungen **PSET** (für Pixel-SET) und **PRESET** (für Pixel-RESET) ausgeführt. Die Anweisung **PSET** *(x,y)* gibt dem Bildpunkt mit den Koordinaten *(x,y)* die aktuelle Vordergrundfarbe. Die Anweisung **PRESET** *(x,y)* gibt dem Bildpunkt mit den Koordinaten *(x,y)* die aktuelle Hintergrundfarbe.

Auf monochromen Monitoren ist die Vordergrundfarbe die für geschriebenen Text verwendete Farbe; sie ist normalerweise weiß, amber oder hellgrün; die Hintergrundfarbe eines monochromen Monitors ist normalerweise schwarz oder dunkelgrün. Mit dem wahlweisen Argument *Farbe* können Sie eine andere Farbe auswählen, mit der **PSET** und **PRESET** arbeiten. Die Syntax ist in diesem Fall:

PSET *(x,y), Farbe*

oder

PRESET *(x,y), Farbe*

Weitere Informationen zur Auswahl von Farben finden Sie in Abschnitt 5.7.

Da **PSET** standardmäßig die aktuelle Vordergrundfarbe und **PRESET** standardmäßig die aktuelle Hintergrundfarbe verwendet, löscht **PRESET** ohne ein Farbargument einen mit **PSET** graphisch dargestellten Punkt, wie im nächsten Beispiel gezeigt:

```
SCREEN 2        ' Auflösung 640 x 200
PRINT "Zum Beenden beliebige Taste drücken."
DO
   ' Zeichne eine horizontale Gerade von links nach rechts:
   FOR X = 0 TO 640
      PSET (X, 100)
   NEXT
   ' Lösche die Gerade von rechts nach links:
   FOR X = 640 TO 0 STEP -1
      PRESET (X, 100)
   NEXT
LOOP UNTIL INKEY$ <> ""
END
```

Es ist zwar möglich, jede beliebige Figur nur mit **PSET**-Anweisungen durch Manipulation einzelner Bildpunkte zu zeichnen, die Ausgabe neigt jedoch dazu, ziemlich langsam zu sein, da die meisten Bilder aus vielen Bildpunkten bestehen. BASIC hat mehrere Anweisungen, die die Geschwindigkeit, mit der einfache Figuren - wie Geraden, Rechtecke und Ellipsen – gezeichnet werden, drastisch erhöhen, wie in Abschnitt 5.3.2 und 5.4.1 bis 5.4.4 gezeigt.

5.3.2 Zeichnen von Geraden und Rechtecken mit Hilfe von LINE

Bei der Verwendung von **PSET** oder **PRESET** geben Sie nur ein Koordinatenpaar an, da Sie nur mit einem Punkt auf dem Bildschirm arbeiten. Mit **LINE** geben Sie zwei Paare an, eines für jedes Ende eines Geradenabschnittes. Die einfachste Form der **LINE**-Anweisung lautet wie folgt:

LINE *(x1,y1)* - *(x2,y2)*

wobei *(x1,y1)* die Koordinaten des einen Endes des Geradenabschnittes und *(x2,y2)* die Koordinaten des anderen Endes darstellen. Folgende Anweisung zeichnet zum Beispiel eine gerade Linie vom Bildpunkt mit den Koordinaten `(10,10)` zu dem Bildpunkt mit den Koordinaten `(150,200)`:

```
SCREEN 1
LINE (10, 10)-(150, 200)
```

Beachten Sie, daß BASIC die Reihenfolge der Koordinatenpaare nicht berücksichtigt: eine Gerade von *(x1,y1)* nach *(x2,y2)* ist dasselbe wie eine Gerade von *(x2,y2)* nach *(x1,y1)*. Daher könnte die vorhergehende Anweisung auch wie folgt geschrieben werden:

```
SCREEN 1
LINE (150, 200)-(10, 10)
```

Die Umkehrung der Koordinatenreihenfolge kann jedoch eine Auswirkung auf nachfolgende Graphikanweisungen haben, wie im nächsten Abschnitt gezeigt.

5.3.2.1 Verwendung der Option STEP

Bis zu diesem Punkt wurden Bildschirmkoordinaten als absolute Werte vorgestellt, die die horizontalen und vertikalen Abstände von der äußerst linken oberen Bildschirmecke angeben, deren Koordinaten `(0,0)` sind. Durch Verwendung der Option **STEP** in jeder der folgenden Graphikanweisungen können jedoch die auf **STEP** folgenden Koordinaten relativ zum letzten Bildschirmpunkt, auf den Bezug genommen wurde, gemacht werden:

CIRCLE	**GET**
LINE	**PAINT**
PRESET	**PSET**
PUT	

Wenn Sie sich Bilder auf dem Bildschirm wie mit einem dünnen Pinsel gezeichnet vorstellen, der genau die Größe eines Bildpunktes hat, dann ist der letzte Punkt, auf den Bezug genommen wird, die Position dieses Pinsels oder "graphischen Cursors" nach Beenden eines Bildes. Zum Beispiel belassen die folgenden Anweisungen den graphischen Cursor auf dem Bildpunkt mit den Koordinaten `(100,150)`:

```
SCREEN 2
LINE (10, 10)-(100, 150)
```

Wenn in diesem Programm die nächste Graphikanweisung

```
PSET STEP (20, 20)
```

lautet, befindet sich der von **PSET** gezeichnete Punkt nicht in dem oberen linken Quadranten des Bildschirms. Stattdessen hat die Option **STEP** die Koordinaten `(20, 20)` relativ zu dem letzten angesprochenen Punkt, mit den Koordinaten `(100, 150)` gemacht. Dies gibt dem Punkt die absoluten Koordinaten `(100+20, 150+20)` oder `(120, 170)`.

Im vorigen Beispiel wird der letzte angesprochene Punkt von einer vorhergehenden Graphikanweisung bestimmt. Ein Bezugspunkt kann auch innerhalb derselben Anweisung bestimmt werden, wie aus dem nächsten Beispiel ersichtlich:

```
' Setze die Auflösung auf 640 x 200 Bildpunkte und mache die
' Bildschirmmitte zum letzten Bezugspunkt:
SCREEN 2

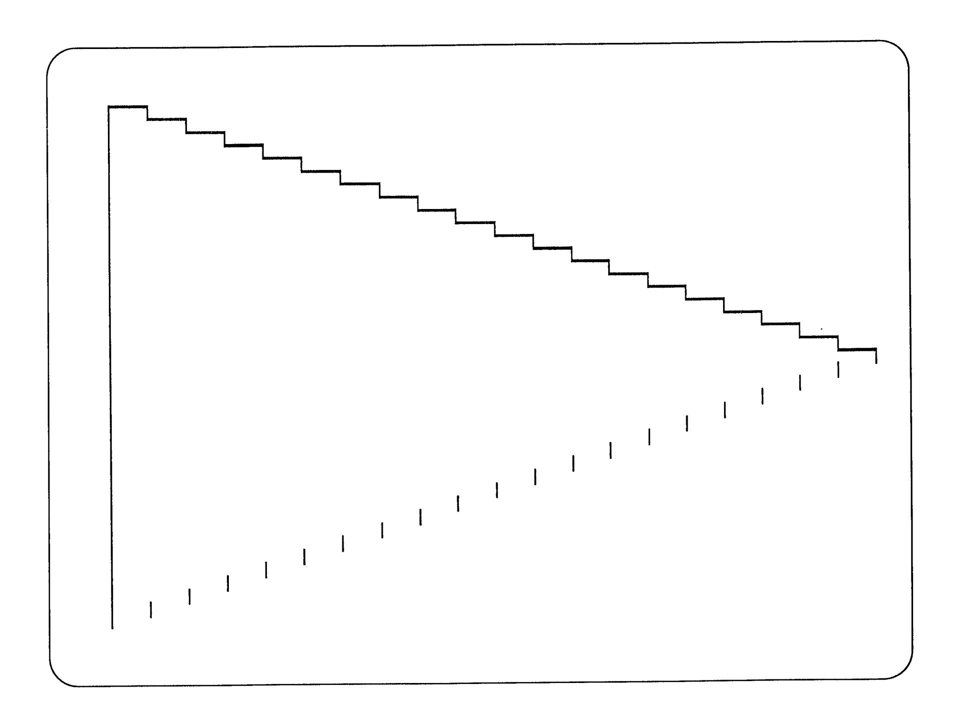
' Zeichne eine Gerade von der linken unteren Ecke des
' Bildschirms zur oberen linken Ecke:
LINE STEP(-310, 100) -STEP(0, -200)

' Zeichne die "Treppenstufen" abwärts von der oberen linken
' Ecke zur rechten Seite des Bildschirms:
FOR I% = 1 TO 20
   LINE -STEP(30, 0)      ' Zeichne die horizontale Gerade.
   LINE -STEP(0, 5)       ' Zeichne die vertikale Gerade.
NEXT

' Zeichne die unverbundenen vertikalen Geradenstücke von der
' rechten Seite zur unteren linken Ecke:
FOR I% = 1 TO 20
   LINE STEP(-30, 0) -STEP(0, 5)
NEXT

DO: LOOP WHILE INKEY$ = ""      ' Warte auf Tastenbetätigung.
```

Ausgabe



Hinweis Beachten Sie die leere **DO**-Schleife am Ende des letzten Programmes. Beim Starten eines kompilierten selbständigen BASIC-Programmes, das graphische Ausgabe erzeugt, benötigt das Programm einen Mechanismus wie diesen am Ende, um die Ausgabe auf dem Bildschirm zu halten. Andernfalls verschwindet diese vom Bildschirm, bevor der Benutzer Zeit hat, sie zu betrachten.

5.3.2.2 Zeichnen von Rechtecken

Mit Hilfe der bereits dargestellten Formen der **LINE**-Anweisung ist es ziemlich einfach, ein kurzes Programm zu schreiben, das vier gerade Linien zur Bildung eines Rechteckes wie folgt verbindet:

```
SCREEN 1   ' Auflösung 320 x 200 Bildpunkte
' Zeichne ein Rechteck, das auf einer Seite 120 Bildpunkte
' mißt:
LINE (50, 50)-(170, 50)
LINE -STEP (0, 120)
LINE -STEP (-120, 0)
LINE -STEP (0, -120)
```

BASIC unterstützt jedoch einen einfacheren Weg, Rechtecke durch Verbinden einer einzelnen **LINE**-Anweisung mit der Option **B** (für Box) zu zeichnen. Die Option **B** wird im nächsten Beispiel, das dieselbe Ausgabe wie die vier **LINE**-Anweisungen im vorhergehenden Programm erzeugt, veranschaulicht:

```
SCREEN 1    ' Auflösung 320 x 200 Bildpunkte
' Zeichne ein Rechteck mit den Koordinaten (50, 50) für die
' obere linke Ecke und (170, 170) für die untere rechte
' Ecke:
LINE (50, 50)-(170, 170), , B
```

Beim Hinzufügen der Option **B** verbindet die **LINE**-Anweisung nicht länger die angegebenen zwei Punkte mit einer geraden Linie; stattdessen zeichnet sie ein Rechteck, dessen gegenüberliegende Ecken (obere linke und untere rechte Ecke) sich an den angegebenen Stellen befinden.

Im letzten Beispiel stehen zwei Kommata vor dem **B**. Das erste Komma funktioniert als Platzhalter für eine unbenutzte Option (das Argument *Farbe*), die das Aussuchen der Farbe einer Zeile oder der Farbe der Rechteckseiten ermöglicht. (Weitere Informationen über die Verwendung von Farben finden Sie in Abschnitt 5.7.)

Wie schon zuvor ist die Reihenfolge, in der die Koordinatenpaare angegeben werden, belanglos, so daß das Rechteck des letzten Beispiels ebenso mit dieser Anweisung gezeichnet werden könnte:

```
LINE (170, 170)-(50, 50), , B
```

Das Hinzufügen der Option **F** (für Füllen) nach **B** malt das Innere des Rechtecks in derselben Farbe wie die Seiten aus. Bei einem monochromen Bildschirm ist diese die gleiche wie die für geschriebenen Text verwendete Vordergrundfarbe. Wenn Ihre Hardware Farbfähigkeiten besitzt, läßt sich diese Farbe mit dem wahlfreien Argument *Farbe* verändern (siehe auch Abschnitt 5.7.1, "Wahl einer Farbe für graphische Ausgaben").

Die hier eingeführte Schreibweise zum Zeichnen eines Rechtecks ist die allgemein in BASIC zur Definierung eines rechtwinkligen Graphikbereiches verwendete Syntax, die ebenfalls in den Anweisungen **GET** und **VIEW** erscheint:

{GET | LINE | VIEW} (*x1,y1*) - (*x2,y2*),...

Hier sind (*x1,y1*) und (*x2,y2*) die Koordinaten der sich diagonal gegenüberliegenden Ecken des Rechtecks (oben links und unten rechts). (Erläuterungen zu **VIEW** finden Sie in Abschnitt 5.5, "Definieren eines graphischen Darstellungsfeldes". Informationen zu **GET** und **PUT** sind Abschnitt 5.10, "Grundlegende Techniken der Animation" zu entnehmen.)

5.3.2.3 Zeichnen von punktierten Geraden

Die vorhergehenden Abschnitte erläutern die Verwendung von **LINE**, um ununterbrochene Geraden zu ziehen und diese für Rechtecke zu verwenden; das heißt, daß keine Bildpunkte ausgelassen werden. Unter Verwendung einer anderen Option mit **LINE** können stattdessen punktierte oder unterbrochene Geraden gezeichnet werden. Dieses Verfahren wird als "Geradengestaltung" bezeichnet. Es folgt die Syntax, mit der unter Verwendung der aktuellen Vordergrundfarbe eine unterbrochene Gerade vom Punkt (*x1,y1*) zum Punkt (*x2,y2*) gezeichnet wird:

LINE (*x1,y1*) - (*x2,y2*) ,,**[B]**, *Struktur*

Hier ist *Struktur* eine 16-Bit dezimale oder hexadezimale Ganzzahl. **LINE** verwendet die binäre Darstellung des Arguments der Geradengestaltung, um Striche und Leerzeichen zu erzeugen, wobei ein 1-Bit "schalte den Bildpunkt ein" und ein 0-Bit "lasse den Bildpunkt ausgeschaltet" bedeutet. Zum Beispiel ist die hexadezimale Ganzzahl &HCCCC gleich der binären Ganzzahl 1100110011001100; bei Verwendung als *Struktur*-Argument wird eine Gerade mit abwechselnd zwei Bildpunkten ein, zwei Bildpunkten aus gezeichnet.

Beispiel

Das folgende Beispiel zeigt unterschiedlich unterbrochene Geraden, die mit verschiedenen Werten für *Struktur* erzeugt werden:

```
SCREEN 2       ' Auflösung 640 x 200 Bildpunkte
' Strukturdaten:
DATA &HCCCC, &HFF00, &HF0F0
DATA &HF000, &H7777, &H8888
Zeile%    = 4
Spalte%   = 4
XLinks%   = 60
XRechts%  = 600
Y%        = 28
FOR I% = 1 TO 6
   READ Struktur%
   LOCATE Zeile%, Spalte%
   PRINT HEX$(Struktur%)
   LINE (XLinks%, Y%)-(XRechts%,Y%),,,Struktur%
   Zeile% = Zeile% + 3
   Y%     = Y% + 24
NEXT
```

Ausgabe

```
CCCC ........................................................................

FF00 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

F0F0 ------------------------------------------------------------------------

F000 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

7777 ------------------------------------------------------------------------

8888 ........................................................................


Beliebige Taste zum Fortsetzen drücken
```

5.4 Zeichnen von Kreisen und Ellipsen mit CIRCLE

Die Anweisung **CIRCLE** zeichnet eine Vielfalt kreisförmiger, elliptischer oder ovaler Formen. Außerdem zeichnet **CIRCLE** Bögen (Kreisausschnitte), sowie Keile, die den Stücken einer Torte gleichen. Im Graphikmodus können fast alle Kurvenarten mit einer Abwandlung von **CIRCLE** erstellt werden.

5.4.1 Zeichnen von Kreisen

Um einen Kreis von Hand zu zeichnen, genügt es, zwei Daten zu kennen: die Position seines Mittelpunktes sowie die Länge seines Radius (der Abstand vom Mittelpunkt zu einem beliebigen Punkt auf dem Kreis). Diese Informationen und eine sichere Hand (oder besser ein Zirkel) genügen zum Zeichnen eines Kreises.

Ebenso benötigt BASIC lediglich die Position des Kreismittelpunktes sowie die Länge des Radius zum Zeichnen eines Kreises. Die einfachste Form der **CIRCLE**-Syntax ist

CIRCLE [STEP] (*x,y*), *Radius*

wobei *x,y* die Koordinaten des Mittelpunktes bilden und *Radius* der Radius des Kreises ist. Das nächste Beispiel zeichnet einen Kreis mit dem Mittelpunkt (200,100) und einem Radius von 75:

```
SCREEN 2
CIRCLE (200, 100), 75
```

Das vorhergehende Beispiel könnte wie folgt umgeschrieben werden, indem die **STEP**-Option die Koordinaten relativ zum Mittelpunkt und nicht zur oberen linken Ecke macht:

```
SCREEN 2        ' Mache Mittelpunkt des Bildschirms (320,100)
                ' zum Bezugspunkt für die CIRCLE-Anweisung
CIRCLE STEP (-120, 0), 75
```

5.4.2 Zeichnen von Ellipsen

Die **CIRCLE**-Anweisung stellt automatisch ein "Seitenverhältnis" (Aspect Ratio) ein, um sicherzustellen, daß der Kreis auf dem Bildschirm rund, nicht ellipsenförmig erscheint. Dennoch müssen Sie vielleicht das Seitenverhältnis anpassen, um Kreise auf Ihrem Bildschirm richtig darzustellen. Das Seitenverhältnis kann auch verändert werden, um eine ovale Figur, die als Ellipse bezeichnet wird, auf dem Bildschirm zu zeichnen. In beiden Fällen ist folgende Syntax zu verwenden:

CIRCLE [STEP] (*x,y*), *Radius* , , , , *Aspekt*

wobei *Aspekt* eine positive reelle Zahl ist. (Weitere Informationen zu dem Seitenverhältnis und seiner Berechnung für verschiedene Bildschirmmodi, finden Sie in Abschnitt 5.4.5.)

Die zusätzlichen Kommata zwischen *Radius* und *Aspekt* sind Platzhalter für andere Optionen, die **CIRCLE** die zu benutzende Farbe, (im Fall eines Farbmonitoradapters und eines farbunterstützenden Bildschirmmodus), oder den zu zeichnenden Bogen oder Keil angeben (weitere Informationen zu diesen Optionen finden Sie in den Abschnitten 5.4.3, "Zeichnen von Bögen", und 5.7.1,"Wahl einer Farbe für graphische Ausgaben").

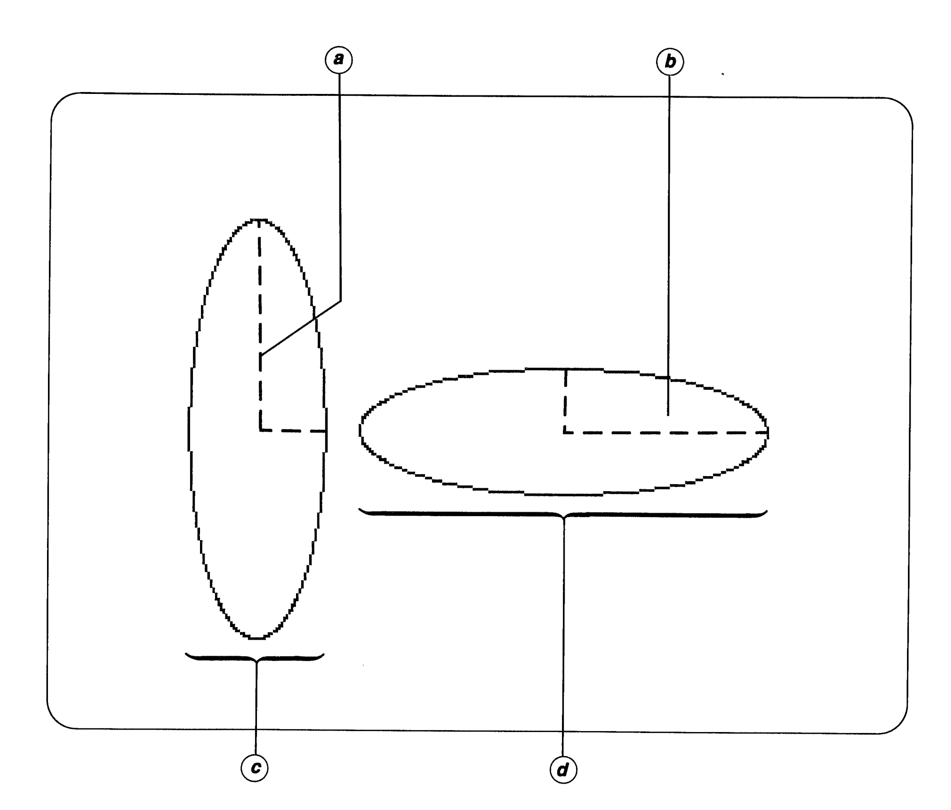
Da das Argument *Aspekt* das Verhältnis der vertikalen zur horizontalen Dimension angibt, führen große Werte für *Aspekt* zu Ellipsen, die sich entlang der vertikalen Achse erstrecken, während kleine Werte für *Aspekt* zu Ellipsen führen, die sich entlang der horizontalen Achse erstrecken. Da eine Ellipse zwei Radien hat – einen horizontalen *x*-Radius und einen vertikalen *y*-Radius – verwendet BASIC das einzelne Argument *Radius* in einer **CIRCLE**-Anweisung wie folgt: wenn *Aspekt* kleiner als 1 ist, ist *Radius* der *x*-Radius; wenn *Aspekt* größer gleich 1 ist, ist *Radius* der *y*-Radius.

Beispiel

Das folgende Beispiel und seine Ausgabe zeigen, wie unterschiedliche Werte von *Aspekt* sich darauf auswirken, ob die **CIRCLE**-Anweisung das Argument *Radius* als den *x*-Radius oder den *y*-Radius einer Ellipse verwendet.

```
SCREEN 1
' Dies zeichnet die linke Ellipse:
CIRCLE (60, 100), 80, , , , 3
' Dies zeichnet die rechte Ellipse:
CIRCLE (180, 100), 80, , , , 3/10
```

Ausgabe



a) Radius
b) Radius
c) In der CIRCLE-Anweisung ist das Seitenverhältnis größer als 1.
d) In der CIRCLE-Anweisung ist das Seitenverhältnis kleiner als 1.

5.4.3 Zeichnen von Bögen

Ein Bogen ist ein Ellipsenausschnitt; mit anderen Worten, eine kurze, gekrümmte Linie. Um zu verstehen, wie die **CIRCLE**-Anweisung einen Bogen zeichnet, müssen Sie wissen, wie BASIC Winkel mißt.

BASIC verwendet das Bogenmaß als Einheit zur Winkelmessung sowohl in der Anweisung **CIRCLE**, als auch in eingebauten trigonometrischen Funktionen wie **COS**, **SIN** oder **TAN**. (Die einzige Ausnahme zu dieser Verwendung des Bogenmaßes ist die Anweisung **DRAW**, die Winkelangaben in Grad erwartet. Weitere Informationen zu **DRAW** finden Sie in Abschnitt 5.9.)

Das Bogenmaß (Einheit rad für Radiant) ist mit dem Radius eines Kreises eng verknüpft. Tatsächlich ist das Wort "Radiant" von dem Wort "Radius" abgeleitet. Der Umfang eines Kreises ist gleich 2*π**Radius*, wobei π ungefähr 3,14159265 beträgt. Ähnlich ist die Anzahl der Radianten in einer kompletten Winkeldrehung (oder 360°) gleich 2*π oder etwas mehr als 6,28. Wenn Sie es gewohnt sind, Winkel in Grad anzugeben, sind untenstehend einige der wichtigsten Umrechnungen zu finden:

Winkel in Grad	***Winkel in Bogenmaß***	
360	2π	(ungefähr 6,283)
180	π	(ungefähr 3,142)
90π	π/2	(ungefähr 1,571)
60	π/3	(ungefähr 1,047)

Wenn Sie sich ein Zifferblatt auf dem Bildschirm vorstellen, mißt **CIRCLE** den Winkel ab der Position "3.00 Uhr", entgegen dem Uhrzeigersinn, wie in Abbildung 5.2 gezeigt.

Abbildung 5.2 Angabe von Winkeln für CIRCLE

Die allgemeine Formel zur Umrechnung von Grad in rad ist die Multiplikation des Gradbetrages mit $\pi/180$.

Zum Zeichnen eines Bogens sind die Winkelargumente anzugeben, die die Grenzen des Bogens definieren:

CIRCLE [STEP] (*x*,*y*) ,*Radius*,**[Farbe]**,*Beginn*,*Ende*[,*Aspekt*]

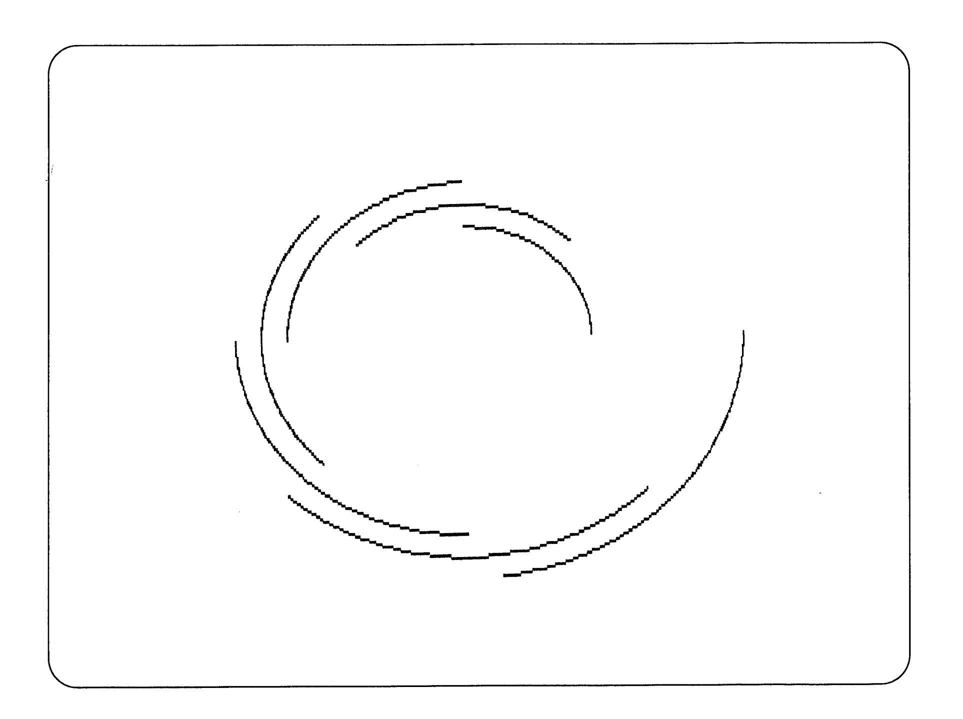
Beispiel

Die **CIRCLE**-Anweisungen des nächsten Beispiels zeichnen sieben Bögen, wobei der innerste Bogen in der Position "3.00 Uhr" (0 Bogenmaß) und der äußerste Bogen in der Position "6.00 Uhr" (3π/2 Bogenmaß) beginnt, wie Sie in der Ausgabe feststellen können:

```
SCREEN 2
CLS
CONST PI = 3.141592653589#      ' Konstante doppelter
                                ' Genauigkeit

StartWinkel = 0
FOR Radius% = 100 TO 220 STEP 20
   EndWinkel = StartWinkel + (PI / 2.01)
   CIRCLE (320, 100), Radius%, , StartWinkel, EndWinkel
   StartWinkel = StartWinkel + (PI / 4)
NEXT Radius%
```

Ausgabe



5.4.4 Zeichnen von Tortendiagrammen

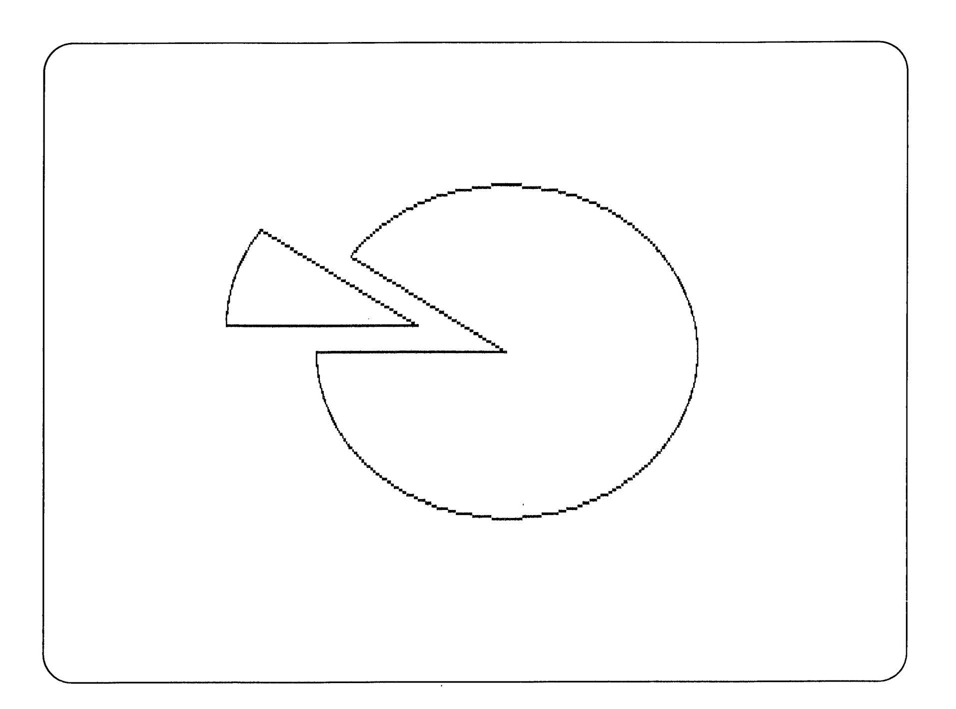
Wird für **CIRCLE** entweder das Argument *Beginn* oder das Argument *Ende* negativ angegeben, kann der Bogen an seinem Anfangs- bzw. Endpunkt mit dem Kreismittelpunkt verbunden werden. Werden beide Argumente negativ angegeben, können Formen gezeichnet werden, die von einem Keil, der einem Stück Torte gleicht, bis zu der Torte selbst (ohne das fehlende Stück) reichen.

Beispiel

Dieses Beispiel zeichnet ein Tortendiagramm, dem ein Stück fehlt.

```
SCREEN 2
CONST RADIUS = 150, PI = 3.141592653589#
StartWinkel = 2.5
EndWinkel = PI
' Zeichne den Keil:
CIRCLE (320, 100), RADIUS, , -StartWinkel, -EndWinkel
' Vertausche die Werte für den Start- und Endwinkel:
SWAP StartWinkel, EndWinkel
' Bewege den Mittelpunkt 10 Bildpunkte nach unten und
' 70 Bildpunkte nach rechts, zeichne dann die "Torte", der
' der Keil fehlt:
CIRCLE STEP(70, 10), RADIUS, , -StartWinkel, -EndWinkel
```

Ausgabe



5.4.5 Zeichnen von Formen, die mit dem Seitenverhältnis anzupassen sind

Wie in Abschnitt 5.4.2, "Zeichnen von Ellipsen", erläutert, korrigiert die BASIC-Anweisung **CIRCLE** automatisch das Seitenverhältnis, das die Skalierung der Figuren auf dem Bildschirm bestimmt. Mit anderen Graphikanweisungen müssen jedoch horizontale und vertikale Dimensionen selbst skaliert werden, um Formen im richtigen Größenverhältnis erscheinen zu lassen. Obwohl die folgende Anweisung zum Beispiel ein Viereck zeichnet, das auf allen Seiten 100 Bildpunkte mißt, sieht es nicht wie ein Quadrat aus:

```
SCREEN 1
LINE (0, 0)-(100, 100), , B
```

Dies ist tatsächlich keine optische Täuschung; das Rechteck ist wirklich höher als es breit ist. Der Grund dafür ist, daß im Bildschirmmodus 1 mehr Platz zwischen vertikalen als zwischen horizontalen Bildpunkten gelassen wird. Um ein richtiges Quadrat zu zeichnen, muß das Seitenverhältnis geändert werden.

Das Seitenverhältnis ist wie folgt definiert: betrachten Sie in einem gegebenen Bildschirmmodus zwei Geraden, eine vertikale und eine horizontale, die gleich lang erscheinen. Das Seitenverhältnis ist die Anzahl von Bildpunkten auf der vertikalen Geraden, dividiert durch die Anzahl der Bildpunkte auf der horizontalen Geraden. Dieses Verhältnis ist von zwei Faktoren abhängig:

1. Aufgrund der Art, in der Bildpunkte auf den meisten Bildschirmen in Abständen angeordnet sind, hat eine horizontale Zeile in allen Bildschirmmodi, außer in den Modi 11 und 12, mehr Bildpunkte als eine vertikale Spalte mit genau der gleichen physikalischen Länge.
2. Der Standardbildschirm eines Personal Computers ist breiter als er hoch ist. Normalerweise ist das Verhältnis von Bildschirmhöhe zu Bildschirmbreite gleich 3:4.

Um festzustellen, wie diese beiden Faktoren bei der Aufstellung des Seitenverhältnisses zusammenwirken, ist ein Bildschirm nach einer Anweisung **SCREEN 1** zu betrachten, die eine Auflösung von 320 x 200 Bildpunkten angibt. Beim Zeichnen eines Rechtecks vom oberen bis zum unteren Bildschirmrand und dreiviertel des Weges von der linken bis zur gegenüberliegenden Bildschirmseite erhalten Sie ein Quadrat, wie in Abbildung 5.3 dargestellt.

Abbildung 5.3 Das Seitenverhältnis im Bildschirmmodus 1

Wie aus dem Diagramm ersichtlich, hat dieses Quadrat eine Höhe von 200 Bildpunkten und eine Breite von 240 Bildpunkten. Das Verhältnis der Quadrathöhe zu seiner Breite (200/240, oder gekürzt 5/6) ist das Seitenverhältnis für diese Bildschirmauflösung. Mit anderen Worten, um ein Quadrat bei einer Auflösung von 320 x 200 zu zeichnen, müssen Sie seine Bildpunkthöhe gleich 5/6 mal seiner Bildpunktbreite angeben, wie in dem nächsten Beispiel gezeigt:

```
SCREEN 1    ' Auflösung 320 x 200 Bildpunkte
' Die Höhe dieses Vierecks beträgt 100 Bildpunkte und die
' Breite beträgt 120 Bildpunkte, wodurch das Verhältnis von
' Höhe zu Breite gleich 100/120 bzw. 5/6 ist. Das Ergebnis
' ist ein Quadrat:
LINE (50, 50) -STEP (120, 100), , B
```

Es folgt die Formel zur Berechnung des Seitenverhältnisses für einen gegebenen Bildschirmmodus:

(4/3) * (*yBildpunkte* / *xBildpunkte*)

wobei *xBildpunkte* * *yBildpunkte* der aktuellen Bildschirmauflösung entspricht. Im Bildschirmmodus 1 gibt diese Formel zum Beispiel das Seitenverhältnis (4/3) * (200/320), oder 5/6 an; im Bildschirmmodus 2 ist das Seitenverhältnis (4/3) * (200/640) oder 5/12.

Wenn bei Ihrem Bildschirm das Verhältnis Breite zu Höhe nicht gleich 4:3 ist, gibt die folgende Formel eine allgemeinere Form zur Berechnung des Seitenverhältnisses an:

(*Bildschirmbreite* / *Bildschirmhöhe*) * (*yBildpunkte* / *xBildpunkte*)

Die **CIRCLE**-Anweisung kann zum Zeichnen von Ellipsen veranlaßt werden, indem die Werte des Argumentes *Aspekt* verändert werden, wie in Abschnitt 5.4.2 gezeigt.

5.5 Definieren eines graphischen Darstellungsfeldes

Die bisher dargestellten Graphikbeispiele verwenden alle den gesamten Bildschirm als Zeichenfläche mit absoluten Koordinatenabständen, die von der äußersten oberen linken Ecke des Bildschirms gemessen werden.

Mit Hilfe der Anweisung **VIEW** kann jedoch auch eine Art Miniaturbildschirm (als "graphisches Darstellungsfeld" bezeichnet) innerhalb der Grenzen des physikalischen Bildschirms definiert werden. Ist ein graphisches Darstellungsfeld einmal definiert, finden alle graphischen Operationen innerhalb dieses Ausschnittes statt. Jede graphische Ausgabe außerhalb der Grenzen des Darstellungsfeldes wird "abgeschnitten"; das heißt, jeder Versuch, einen Punkt außerhalb des Darstellungsfeldes zu zeichnen, wird ignoriert. Die Verwendung eines Darstellungsfeldes hat zwei Hauptvorteile:

1. Ein Darstellungsfeld erleichtert die Veränderung der Größe und Position eines Bildschirmbereiches, in dem Graphiken erscheinen.
2. Anhand von **CLS 1** kann der Bildschirm innerhalb eines Darstellungsfeldes gelöscht werden, ohne den Bildschirm außerhalb des Darstellungsfeldes zu beeinträchtigen.

Hinweis Um zu lernen, wie ein "Text-Darstellungsfeld" für Ausgaben, die auf den Bildschirm geschrieben werden, angelegt wird, lesen Sie bitte Abschnitt 3.1.6.

Die allgemeine Syntax für **VIEW** (es ist zu beachten, daß die Option **STEP** mit **VIEW** nicht erlaubt ist) lautet

VIEW [[SCREEN] *(x1,y1)* - *(x2,y2)* **[,[*Farbe*],[*Rand*]]]**

wobei *(x1,y1)* und *(x2,y2)* anhand der BASIC-Standardschreibweise für Rechtecke (weitere Informationen finden Sie in Abschnitt 5.3.2.2, "Zeichnen von Rechtecken") die Ecken des Darstellungsfeldes definieren. Die wahlweisen Argumente *Farbe* und *Rand* erlauben Ihnen die Wahl einer Farbe für das Innere bzw. für die Ränder des Darstellungsfeld-Rechteckes. Weitere Informationen zum Setzen und Verändern von Farben finden Sie in Abschnitt 5.7.

Die **VIEW**-Anweisung ohne Argumente läßt den gesamten Bildschirm zum Darstellungsfeld werden. Die **VIEW**-Anweisung ohne die Option **SCREEN** läßt alle Bildpunktkoordinaten relativ zum Darstellungsfeld, nicht zum gesamten Bildschirm, werden. Mit anderen Worten ist nach der Anweisung

```
VIEW (50, 60) - (150, 175)
```

der mit

```
PSET (10, 10)
```

gezeichnete Bildpunkt sichtbar, da er sich 10 Bildpunkte unter und 10 Bildpunkte rechts der oberen linken Ecke des Darstellungsfeldes befindet. Beachten Sie, daß die Bildpunkte somit die absoluten Bildschirmkoordinaten (`50+10, 60+10`) oder (`60, 70`) erhalten.

Im Gegensatz dazu beläßt die **VIEW**-Anweisung ohne die **SCREEN**-Option alle Koordinaten absolut; das heißt, Koordinaten werden von den Seiten des Bildschirms, nicht von den Seiten des Darstellungsfeldes gemessen. Daher ist nach der Anweisung

```
VIEW SCREEN (50, 60) - (150, 175)
```

der mit

```
PSET (10, 10)
```

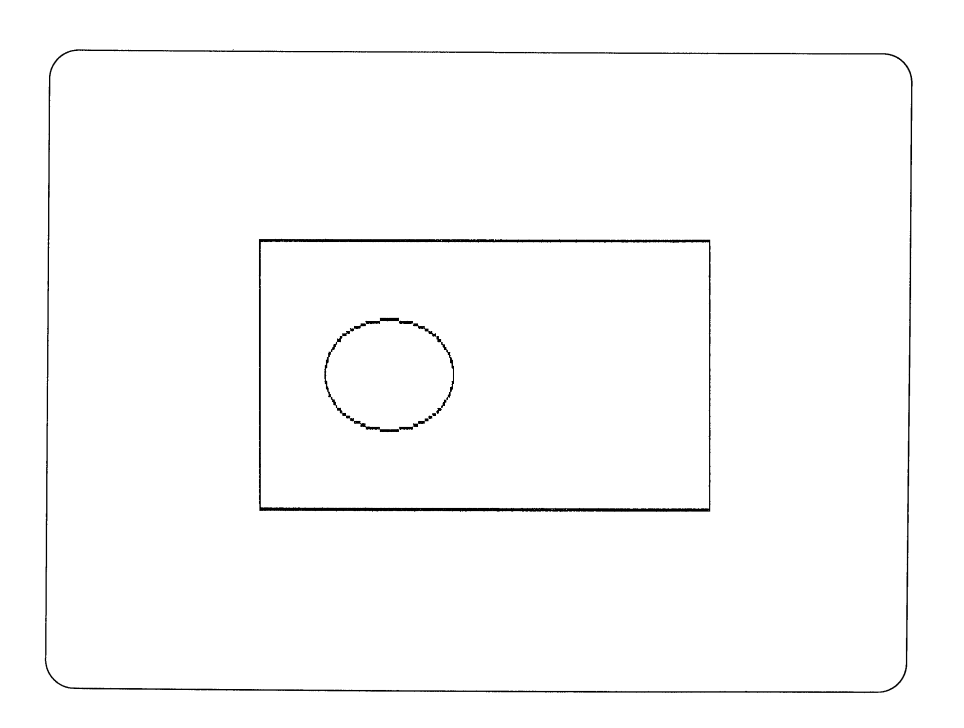
gezeichnete Bildpunkt nicht sichtbar, da er sich 10 Bildpunkte unter und 10 Bildpunkte rechts der oberen linken Ecke des Bildschirms und somit außerhalb des Darstellungsfeldes befindet.

Beispiele

Die Ausgaben der nächsten zwei Beispiele sollen den Unterschied zwischen **VIEW** und **VIEW SCREEN** weiter verdeutlichen:

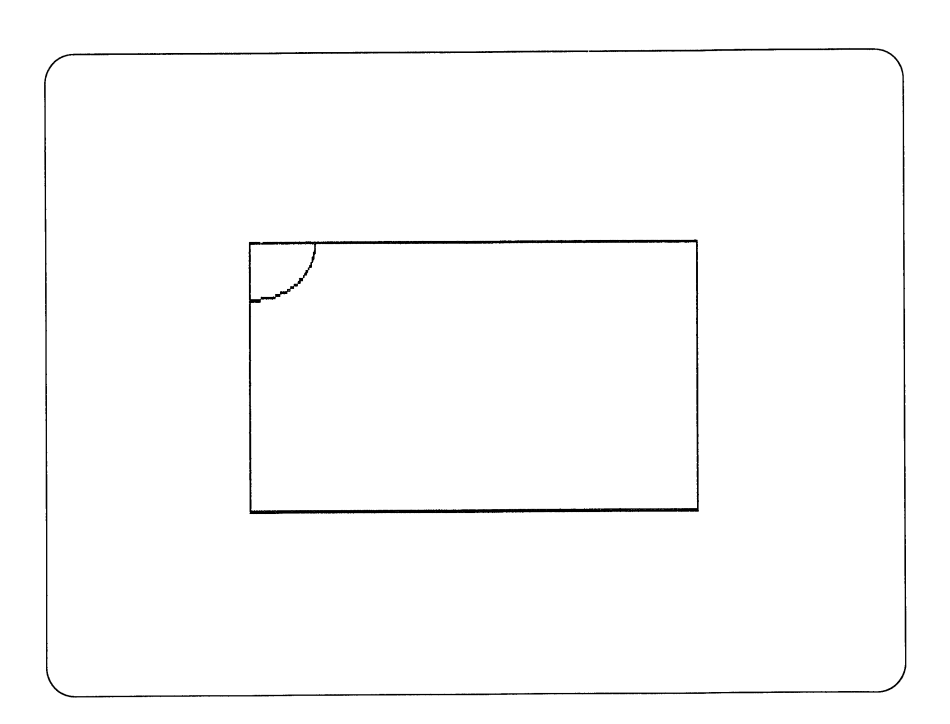
```
SCREEN 2
VIEW (100, 50)-(450, 150), , 1
' Dieser Kreismittelpunkt hat die absoluten Koordinaten
' (100 + 100, 50 + 50) bzw. (200, 100):
CIRCLE (100, 50), 50
```

Ausgabe mit Hilfe von VIEW



```
SCREEN 2
' Dieser Kreismittelpunkt hat die absoluten Koordinaten
' (100, 50), so daß nur ein Teil des Kreises innerhalb des
' Darstellungsfeldes erscheint:
VIEW SCREEN (100, 50)-(450, 150), , 1
CIRCLE (100, 50), 50
```

Ausgabe mit VIEW SCREEN



Beachten Sie, daß eine graphische Ausgabe, die sich außerhalb des aktuellen Darstellungsfeldes befindet, von den Feldseiten abgeschnitten wird und nicht auf dem Bildschirm erscheint.

5.6 Neu Definieren von Koordinaten eines Darstellungsfeldes anhand von WINDOW

Dieser Abschnitt zeigt, wie die Anweisung **WINDOW** und das eigene Koordinatensystem verwendet werden kann, um Bildschirmkoordinaten erneut zu definieren.

In den Abschnitten 5.2 bis 5.5 stellen die zur Positionierung der Bildschirmpunkte auf dem Bildschirm verwendeten Koordinaten physikalische Abstände von der oberen linken Ecke des Bildschirms (oder der oberen linken Ecke des aktuellen Darstellungsfeldes, wenn ein solches mit einer **VIEW**-Anweisung definiert wurde) dar und werden als "physikalische Koordinaten" bezeichnet. Der "Ursprung", oder Referenzpunkt, für physikalische Koordinaten ist immer die obere linke Ecke des Bildschirms oder des Darstellungsfeldes und hat die Koordinaten `(0, 0)`.

Wenn Sie sich auf dem Bildschirm nach unten und nach rechts bewegen, werden die *x*-Werte (horizontale Koordinaten) und *y*-Werte (vertikale Koordinaten) größer, wie in dem oberen Diagramm der Abbildung 5.4 gezeigt. Während dieses System für Bildschirme standardmäßig ist, mag es Ihnen ungewohnt erscheinen, wenn Sie zum Zeichnen von Kurven andere Koordinaten verwendet haben. Zum Beispiel werden *y*-Werte einer Kurve in dem in der Mathematik verwendeten Kartesischen Koordinatensystem nach oben hin größer und nach unten hin kleiner.

Mit Hilfe der BASIC-Anweisung **WINDOW** kann die Art, in der Bildpunkte adressiert werden, verändert und jedes gewünschte Koordinatensystem verwendet werden, wodurch die Einschränkung, physikalische Koordinaten benutzen zu müssen, ausfällt.

Die allgemeine Syntax für **WINDOW** ist

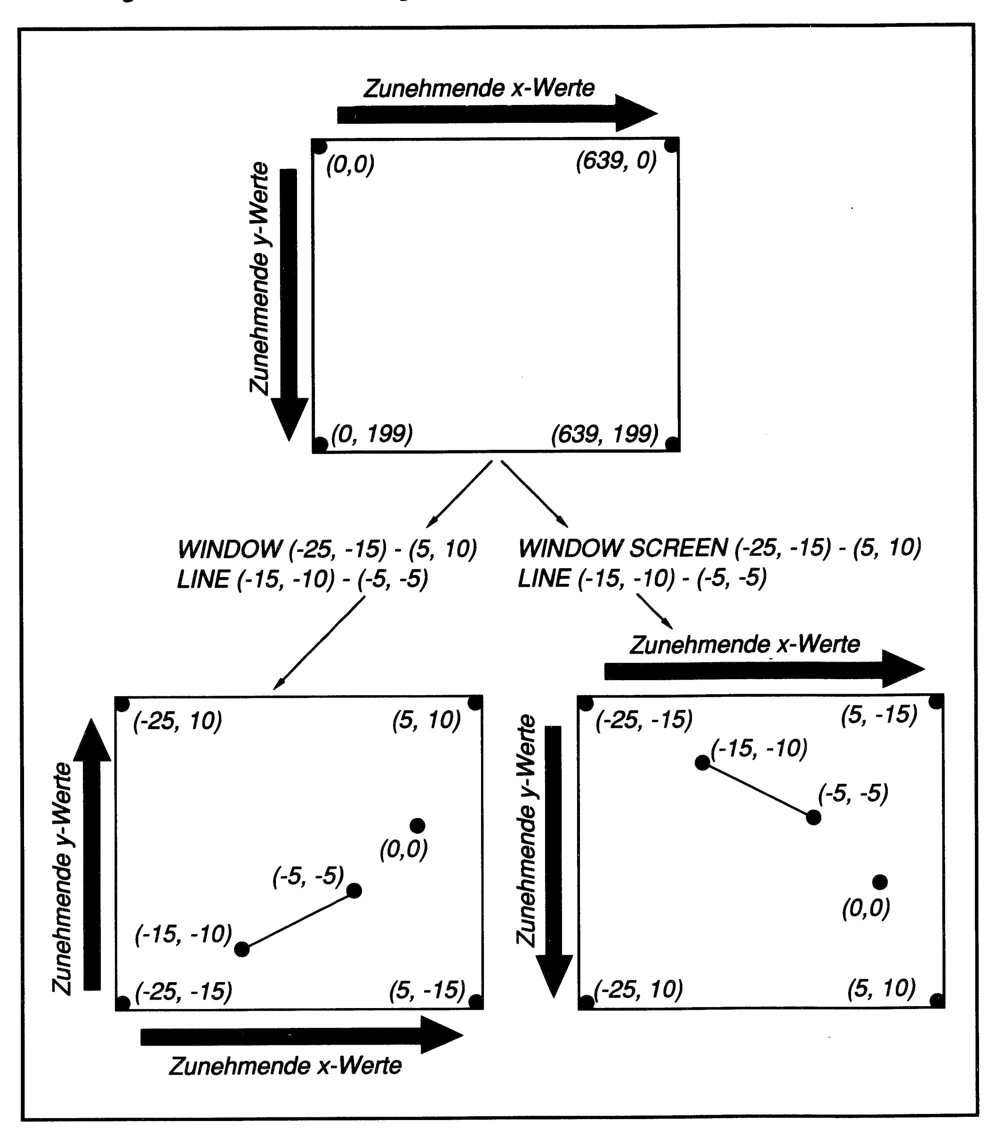
WINDOW [[SCREEN] (*x1,y1*) - (*x2,y2*)]

wobei *y1, y2, x1* und *x2* reelle Zahlen sind, die die obere, untere, linke und rechte Seite des Fensters angeben. Diese Zahlen werden als "logische Koordinaten" bezeichnet. Zum Beispiel teilt die folgende Anweisung Ihren Bildschirm neu ein, so daß er oben und unten von den Geraden `y = 10` bzw. `y = -15` und links und rechts von den Geraden `x = -25` bzw.`x = 5` begrenzt ist:

```
WINDOW (-25, -15)-(5, 10)
```

Nach einer **WINDOW**-Anweisung werden *y*-Werte zum oberen Ende des Bildschirms hin größer. Im Gegensatz dazu werden *y*-Werte nach einer Anweisung **WINDOW SCREEN** zum unteren Ende des Bildschirms hin größer. Die untere Hälfte der Abbildung 5.4 zeigt die Auswirkungen sowohl einer **WINDOW**- als auch einer **WINDOW SCREEN**-Anweisung auf eine im Bildschirmmodus 2 gezogene Gerade. Beachten Sie auch, wie diese beiden Anweisungen die Koordinaten der Bildschirmecken verändern. Eine **WINDOW**-Anweisung ohne Argumente stellt das reguläre physikalische Koordinatensystem wieder her.

Abbildung 5.4 WINDOW im Vergleich zu WINDOW SCREEN

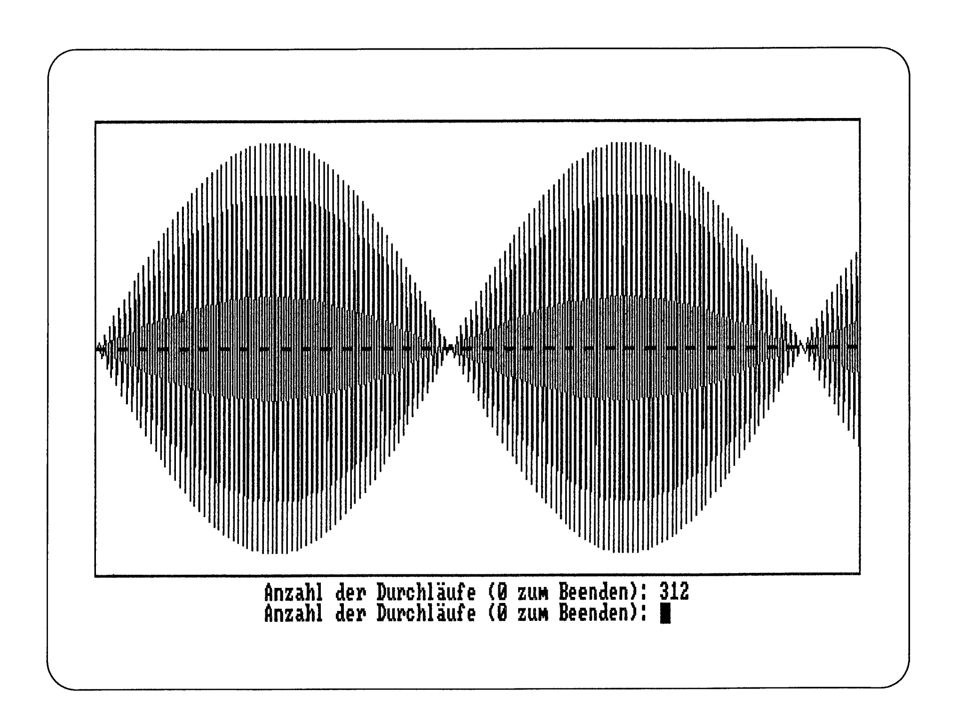


Beispiel

Das folgende Beispiel verwendet sowohl **VIEW** als auch **WINDOW**, um das Schreiben eines Programmes zu vereinfachen, das die Sinus-Funktion für Winkelwerte von 0 Bogenmaß bis π Bogenmaß (oder 0° bis 180°) zeichnet. Dieses Programm befindet sich auf den Originaldisketten in der Datei mit dem Namen *sinus.bas*.

```
SCREEN 2
' Die Größe des Darstellungsfeldes wird passend
' zur zu zeichnenden Kurve gewählt:
VIEW (20, 2)-(620, 172), , 1
CONST PI = 3.141592653589#
' Mache Fenster genügend groß, um Sinuskurve von
' 0 Bogenmaß bis pi Bogenmaß zu zeichnen:
WINDOW (0, -1.1)-(PI, 1.1)
Struktur% = &HFF00      ' Für unterbrochene Gerade.
VIEW PRINT 23 TO 24     ' Rolle geschriebene Ausgabe
                        ' in Zeile 23 und 24.
DO
   PRINT TAB(20);
   PRINT "Anzahl der Durchläufe ";
   INPUT "(0 zum Beenden): ", Durchlauf
   CLS
   LINE (PI, 0)-(0, 0),,,Struktur% 'Zeichne die (hori-
                                   'zontale) x-Achse.
   IF Durchlauf > 0 THEN
      ' Beginn bei (0,0) und zeichne die Kurve:
      FOR X = 0 TO PI STEP .01
          Y = SIN(Durchlauf * X)    ' Berechne die y-
                                    ' Koordinate.
          LINE -(X, Y)  ' Zeichne eine Gerade vom
                        ' letzten Punkt zu dem neuen
                        ' Punkt.
      NEXT X
   END IF
LOOP WHILE Durchlauf > 0
```

Ausgabe



5.6.1 Die Reihenfolge der Koordinatenpaare

Genau wie bei den anderen Graphikanweisungen von BASIC, die rechteckige Bereiche definieren (**GET**, **LINE** und **VIEW**), spielt die Reihenfolge der Koordinatenpaare in einer **WINDOW**-Anweisung keine Rolle. Daher hat das erste der unteren Anweisungspaare dieselbe Auswirkung wie das zweite:

```
VIEW (100, 20)-(300, 120)
WINDOW (-4, -3)-(0, 0)

VIEW (300, 120) - (100, 20)
WINDOW (0, 0)-(-4, -3)
```

5.6.2 Verfolgen der logischen und physikalischen Koordinaten

Die Funktionen **PMAP** und **POINT** sind beim Verfolgen der physikalischen und logischen Koordinaten hilfreich. **POINT**(*Zahl*) gibt die aktuelle Position des graphischen Cursors an, indem es entweder die physikalischen oder logischen Koordinaten (je nach dem Wert von *Zahl*) des letzten Punktes zurückgibt, auf den in einer Graphikanweisung Bezug genommen wurde. **PMAP** ermöglicht die Übertragung von physikalischen in logische Koordinaten und umgekehrt. Die von **PMAP** angegebenen physikalischen Koordinaten sind immer relativ zu dem aktuellen Darstellungsfeld.

Beispiele

Das folgende Beispiel zeigt die unterschiedlichen Werte, die von **POINT**(*Zahl*) für *Zahl*-Werte von 0, 1, 2 oder 3 angegeben werden:

```
SCREEN 2
' Definiere das Fenster für logische Koordinaten:
WINDOW (-10, -30)-(-5, -10)
' Zeichne eine Gerade von dem Punkt mit den logischen
' Koordinaten (-9,-28) zu dem Punkt mit den logischen
' Koordinaten (-6,-24):
LINE (-9, -28)-(-6, -24)
PRINT "Physikalische x-Koordinate des letzten Punktes = ";
PRINT POINT(0)
PRINT "Physikalische y-Koordinate des letzten Punktes = ";
PRINT POINT(1)
PRINT "Logische x-Koordinate des letzten Punktes = ";
PRINT POINT(2)
PRINT "Logische y-Koordinate des letzten Punktes = ";
PRINT POINT(3)
END
```

Ausgabe

```
Physikalische x-Koordinate des letzten Punktes = 511
Physikalische y-Koordinate des letzten Punktes = 139
Logische x-Koordinate des letzten Punktes = -6
Logische y-Koordinate des letzten Punktes = -24
```

Mit der im vorhergehenden Beispiel angegebenen **WINDOW**-Anweisung würden die nächsten vier **PMAP**-Anweisungen die Ausgabe wie folgt schreiben:

```
' Bilde die logische x-Koordinate -6 auf die physikalische
' x-Koordinate ab und gib sie aus:
PhysX% = PMAP(-6, 0)
PRINT PhysX%
' Bilde die logische y-Koordinate -24 auf die physikalische
' y-Koordinate ab und gib sie aus:
PhysY% = PMAP(-24, 1)
PRINT PhysY%
' Bilde physikalisches x wieder auf logisches x ab und gib
' sie aus:
LogischX% = PMAP (PhysX%, 2)
PRINT LogischX%
' Bilde physikalisches y wieder auf logisches y ab und gib
' sie aus:
LogischY% = PMAP (PhysY%, 3)
PRINT LogischY%
```

Ausgabe

```
 511
 139
-6
-24
```

5.7 Die Verwendung von Farben

Arbeiten Sie mit einem Color Graphics Adapter (CGA), können Sie nur zwischen den beiden folgenden Graphikmodi wählen:

1. **SCREEN 2** hat eine hohe Auflösung von 640 x 200 Bildpunkten mit nur einer Vorder- und einer Hintergrundfarbe. Dies wird als "monochrom" bezeichnet, da alle graphischen Ausgaben die gleiche Farbe haben.
2. **SCREEN 1** hat eine Auflösung von 320 x 200 Bildpunkten mit 4 Vorder- und 16 Hintergrundfarben.

Es besteht somit ein Zusammenhang zwischen Farbe und Schärfe in den beiden Bildschirmmodi, die von der meisten Anzeige-/Adapter-Hardware unterstützt werden. Je nach den graphischen Fähigkeiten des Systems brauchen Sie nicht auf die Schärfe zu verzichten, um den vollen Farbbereich zu erhalten. Dieser Abschnitt behandelt hauptsächlich die Bildschirmmodi 1 und 2.

5.7.1 Wahl einer Farbe für graphische Ausgaben

Folgende Liste zeigt den Einsatz des Argumentes *Farbe* in den in den vorhergehenden Abschnitten dieses Kapitels erläuterten graphischen Anweisungen. Diese Liste stellt ebenfalls weitere Optionen (wie z.B. **BF** mit der Anweisung **LINE** oder *Rand* mit der Anweisung **VIEW**) vor, die unterschiedliche Farben aufweisen können. (Es ist zu beachten, daß für diese Anweisungen nicht die komplette Syntax angegeben ist. Diese Zusammenfassung ist nur dazu bestimmt, den Einsatz der Option *Farbe* in Anweisungen, die diese Option akzeptieren, anzugeben).

PSET *(x,y)*, *Farbe*
PRESET *(x,y)*, *Farbe*
LINE *(x1,y1)* - *(x2,y2)*, *Farbe* **[,B[F]]**
CIRCLE *(x,y)*, *Radius*, *Farbe*
VIEW *(x1,y1)* - *(x2,y2)*, *Farbe*, *Rand*

Im Bildschirmmodus 1 ist das Argument *Farbe* ein numerischer Ausdruck mit den Werten 0, 1, 2 oder 3. Jeder dieser Werte, die als "Attribute" bezeichnet werden, stellt eine andere Farbe dar, wie aus folgendem Beispiel ersichtlich:

```
' Zeichne eine "unsichtbare" Gerade (in der Farbe des
' Hintergrundes):
LINE (10, 10)-(310, 10), 0

' Zeichne eine hellblaue (kobaltblaue) Gerade:
LINE (10, 30)-(310, 30), 1

' Zeichne eine purpurrote (violette) Gerade:
LINE (10, 50)-(310, 50), 2

' Zeichne eine weiße Gerade:
LINE (10, 70)-(310, 70), 3
END
```

Wie in den Kommentaren des vorhergehenden Beispiels erwähnt, ergibt der Wert 0 für *Farbe* keine sichtbare Ausgabe, da sie immer in der aktuellen Hintergrundfarbe ist. Zunächst scheint dies kein sinnvoller Farbwert zu sein, dennoch ist er hilfreich beim Entfernen von Abbildungen, ohne dabei den gesamten Bildschirm oder das gesamte Darstellungsfeld zu löschen, wie im nächsten Beispiel gezeigt:

```
SCREEN 1
CIRCLE (100, 100), 80, 2, , , 3      ' Zeichne eine Ellipse.
Pause$ = INPUT$(1)                   ' Warte auf eine
                                     ' Tastenbetätigung.
CIRCLE (100, 100), 80, 0, , , 3      ' Entferne die Ellipse.
```

5.7.2 Verändern der Vorder- oder Hintergrundfarbe

Im oben aufgeführten Abschnitt 5.7.1 wurde der Gebrauch von 4 unterschiedlichen Vordergrundfarben für graphische Ausgabe bereits erklärt. Im Bildschirmmodus 1 gibt es eine größere Vielfalt von insgesamt 16 Farben für die Hintergrundfarbe des Bildschirms.

Zusätzlich kann die Vordergrundfarbe mit Hilfe einer unterschiedlichen "Palette" verändert werden. Im Bildschirmmodus 1 gibt es zwei Paletten oder Gruppen mit vier Farben. Jede Palette weist demselben Attribut eine unterschiedliche Farbe zu; zum Beispiel ist in Palette 1 (die Standardeinstellung) die mit dem Attribut 2 verknüpfte Farbe violett, während in Palette 0 die mit dem Attribut 2 verknüpfte Farbe rot ist. Wenn Sie über eine CGA-Karte verfügen, sind diese Farben für jede Palette festgelegt; das heißt, die in Palette 1 der Nummer 2 zugewiesene Farbe ist immer violett, während die in Palette 0 mit Nummer 2 verknüpfte Farbe immer rot ist.

Wenn Sie einen Enhanced Graphics Adapter (EGA) oder einen Video Graphics Adapter (VGA) haben, können Sie die von jedem Attribut angezeigte Farbe anhand der Anweisung **PALETTE** auswählen. Durch Veränderung der Argumente in der Anweisung **PALETTE** könnten Sie zum Beispiel die mit dem Attribut 1 angezeigte Farbe einmal grün und das nächste mal braun werden lassen. (Weitere Informationen über die Zuweisung von Farben finden Sie in Abschnitt 5.7.3, "Verändern der Farben anhand von PALETTE und PALETTE USING" in diesem Handbuch.)

Im Bildschirmmodus 1 gewährleistet die **COLOR**-Anweisung die Steuerung sowohl der Hintergrundfarbe als auch der Palette der Vordergrundfarben. Es folgt die Syntax für **COLOR** im Bildschirmmodus 1:

COLOR [*Hintergrund*] [,*Palette*]

Das Argument *Hintergrund* ist ein numerischer Ausdruck von 0 - 15 und *Palette* ist ein numerischer Ausdruck, der entweder 0 oder 1 sein kann.

Tabelle 5.1 listet die Farben, die durch die 4 verschiedenen Vordergrundnummern in jeder der zwei Paletten erzeugt werden, während Tabelle 5.2 die mit 16 verschiedenen Hintergrundnummern erzeugten Farben aufführt.

Tabelle 5.1 Farbpaletten im Bildschirmmodus 1

Nummer der Vordergrundfarbe	***Farbe in Palette 0***	***Farbe in Palette 1***
0	aktuelle hintergrundfarbe	aktuelle Hintergrundfarbe
1	grün	kobaltblau (blau-grün)
2	rot	violett (hellpurpur)
3	braun	weiß (auf manchen Bildschirmen hellgrau)

Tabelle 5.2 Hintergrundfarben im Bildschirmmodus 1

Nummer der Hintergrundfarbe	***Farbe***
0	schwarz
1	blau
2	grün
3	kobaltblau
4	rot
5	violett
6	braun (auf manchen Bildschirmen dunkelgelb)
7	weiß (auf manchen Bildschirmen hellgrau)
8	dunkelgrau (auf manchen Bildschirmen schwarz)
9	hellblau
10	hellgrün
11	hellkobaltblau
12	hellrot
13	hellviolett
14	hellgelb (auf manchen Bildschirmen eventuell mit einem grünlichen Ton)
15	leuchtend weiß oder sehr helles Grau

Beispiel

Das folgende Programm zeigt alle Kombinationen der 2 Farbpaletten mit den 16 unterschiedlichen Hintergrund-Bildschirmfarben. Dieses Programm befindet sich in einer Datei mit dem Namen *farben.bas* auf den QuickBASIC-Originaldisketten.

```
SCREEN 1
Esc$ = CHR$(27)
' Zeichne drei Rechtecke und male das Innere jedes
' Rechtecks mit einer anderen Farbe aus:
FOR FarbWert = 1 TO 3
   LINE (X, Y)-STEP(60, 50), FarbWert, BF
   X = X + 61
   Y = Y + 51
NEXT FarbWert
LOCATE 21, 1
PRINT "ESC zum Beenden."
PRINT "Weiter mit jeder Taste."
' Begrenze zusätzlich geschriebene Ausgabe auf die
' 23. Zeile:
VIEW PRINT 23 TO 23
DO
   PaletteWert = 1
   DO
      ' PaletteWert ist entweder 1 oder 0:
      PaletteWert = 1 - PaletteWert
      ' Setze die Hintergrundfarbe und wähle die
      ' Palette:
      COLOR HintGrundWert, PaletteWert
      PRINT "Hintergrund ="; HintGrundWert;
      PRINT "Palette ="; PaletteWert;
      Pause$ = INPUT$(1)   ' Warte auf eine
                           ' Tastenbetätigung.
      PRINT
   ' Verlasse die Schleife, wenn beide Paletten
   ' gezeigt wurden, oder wenn der Benutzer die ESC-
   ' Taste betätigt hat:
   LOOP UNTIL PaletteWert = 1 OR Pause$ = Esc$
   HintGrundWert = HintGrundWert + 1
```

```
' Verlasse diese Schleife, wenn alle 16
' Hintergrundfarben gezeigt wurden, oder wenn der
' Benutzer die ESC-Taste betätigt hat:
LOOP UNTIL HintGrundWert > 15 OR Pause$ = Esc$

SCREEN 0          ' Stelle den Textmodus und 80-Spalten
WIDTH 80          ' Bildschirmbreite wieder her.
```

5.7.3 Verändern der Farben anhand von PALETTE und PALETTE USING

Der vorhergehende Abschnitt zeigte, wie Sie die von einem Attribut angezeigte Farbe einfach durch Festlegung einer anderen Palette in der **COLOR**-Anweisung verändern können. Dadurch sind Sie jedoch auf zwei feste Farbpaletten mit nur jeweils vier Farben angewiesen. Darüberhinaus kann jedes Attribut nur für eine der zwei möglichen Farben stehen; das Attribut 1 kann zum Beispiel nur grün oder kobaltblau festlegen.

Wenn Sie dagegen über eine EGA oder eine VGA-Graphikkarte verfügen, sind Ihre potentiellen Auswahlmöglichkeiten erheblich größer. (Sollte dies nicht der Fall sein, brauchen Sie diesen Abschnitt nicht durchzulesen.) Sie können beispielsweise je nach Videospeicherkapazität des Computers mit einer VGA-Karte bis zu 256K an Farben einer Palette wählen (es sind tatsächlich über 256.000) und diese Farben 256 verschiedenen Attributen zuweisen. Sogar eine EGA-Karte ermöglicht die Anzeige von 16 unterschiedlichen Farben aus einer Palette mit 64 Farben.

Im Gegensatz zu der **COLOR**-Anweisung gewährleisten die Anweisungen **PALETTE** bzw. **PALETTE USING** eine erheblich größere Flexibilität in der Veränderung der verfügbaren Farbpalette. Mit Hilfe dieser Anweisungen können Sie jede Farbe der Palette jedem Attribut zuweisen. Zum Beispiel erscheint nach der folgenden Anweisung die Ausgabe aller Graphikanweisungen, die Attribut 4 verwenden, in hellviolett (Farbe 13):

```
PALETTE 4, 13
```

Diese Farbveränderung ist unverzüglich und wirkt sich nicht nur auf die nachfolgenden Graphikanweisungen, sondern auch auf jede bereits auf dem Bildschirm befindliche Ausgabe aus. Mit anderen Worten können Sie auf den Bildschirm zeichnen und malen und anschließend die Palette umschalten, um einen sofortigen Farbwechsel wie folgt durchzuführen:

```
SCREEN 8
LINE (50, 50)-(150, 150), 4    ' Zeichne eine Gerade in rot.
SLEEP 1                        ' Programm ruht.
PALETTE 4, 13     ' Attribut 4 entspricht jetzt
                  ' Farbe 13, so daß die in der letzten
                  ' Anweisung gezeichnete Gerade jetzt
                  ' hellviolett ist.
```

Mit Hilfe der Option **USING** der Anweisung **PALETTE** können die den Attributen zugewiesenen Farben alle gleichzeitig geändert werden.

Beispiel

Das nächste Beispiel benutzt die Anweisung **PALETTE USING**, um den Eindruck von Bewegung auf dem Bildschirm zu vermitteln, indem die mit den Attributen 1 bis 15 angezeigten Farben ständig umlaufen.

Dieses Programm befindet sich in der Datei mit dem Namen *palette.bas* auf den QuickBASIC-Originaldisketten.

```
DECLARE SUB InitPalette ()
DECLARE SUB WechselPalette ()
DECLARE SUB ZeichneEllipsen ()

DEFINT A-Z
DIM SHARED PaletteFeld(15)

SCREEN 8                ' Auflösung 640 x 200; 16 Farben

InitPalette             ' Initialisiere PaletteFeld.
ZeichneEllipsen         ' Zeichne und male konzentrische
                        ' Ellipsen.

DO                      ' Schalte die Palette um, bis eine
                        ' Taste betätigt wird.
   WechselPalette
LOOP WHILE INKEY$ = ""

END
'
' =================== InitPalette ==========================
'    Diese Prozedur initialisiert das zum Wechseln der
'    Palette verwendete ganzzahlige Datenfeld.
' ==========================================================
'
SUB InitPalette STATIC
   FOR I = 0 TO 15
      PaletteFeld(I) = I
   NEXT I
END SUB
'
' ================= ZeichneEllipsen =======================
'    Diese Prozedur zeichnet 15 konzentrische Ellipsen
'    und malt das Innere einer jeden mit einer anderen
'    Farbe aus.
' ==========================================================
'
SUB ZeichneEllipsen STATIC
   CONST ASPEKT = 1 / 3
```

```
   FOR FarbWert = 15 TO 1 STEP -1
      Radius = 20 * FarbWert
      CIRCLE (320, 100), Radius, FarbWert, , , ASPEKT
      PAINT (320, 100), FarbWert
   NEXT
END SUB
'
' ================= WechselPalette ===================
'    Diese Prozedur läßt die Palette bei jedem Aufruf
'    um eins rotieren. Z. B. ist nach dem ersten
'    Aufruf von WechselPalette PaletteFeld(1) = 2,
'    PaletteFeld(2) = 3, . . . , PaletteFeld(14) = 15
'    und PaletteFeld(15) = 0
' ====================================================
'
SUB WechselPalette STATIC
   FOR I = 1 TO 15
      PaletteFeld(I) = (PaletteFeld(I) MOD 15) + 1
   NEXT I
   PALETTE USING PaletteFeld(0) ' Schalte die von
                                ' jedem der 15
                                ' Attribute angezeigte
                                ' Farbe um.
END SUB
```

5.8 Ausmalen von Formen

Der obenstehende Abschnitt 5.3.2.2 erklärt, wie ein Rechteck mit der Option **B** der **LINE**-Anweisung gezeichnet und dann durch Hinzufügen der Option **F** (für Füllen) ausgemalt wird:

```
SCREEN 1
' Zeichne ein Quadrat, male das Innere dann mit Farbe 1 aus
' (Kobaltblau in der Standard-Palette):
LINE (50, 50)-(110, 100), 1, BF
```

Mit der BASIC-Anweisung **PAINT** kann jede geschlossene Figur mit der jeweils gewünschten Farbe gefüllt werden. Anhand von **PAINT** lassen sich ebenfalls geschlossene Figuren mit den eigenen gewohnten Mustern wie Streifen und Karos ausfüllen, wie in Abschnitt 5.8.2, "Zeichnen mit Mustern: Ausfüllen mit Mustern" gezeigt.

5.8.1 Ausmalen mit Farben

Um eine geschlossene Form mit einer durchgehenden Farbe auszumalen, ist nachstehende Form der Anweisung **PAINT** anzuwenden:

PAINT [STEP] (*x,y*) **[,[*Innen*],[*Rand*]]**

Hier sind *x,y* die Koordinaten eines Punktes im Inneren der auszumalenden Abbildung; *Innen* ist die Nummer der gewünschten Farbe und *Rand* die Farbnummer der Figurenkontur.

Zum Beispiel zeichnen folgende Programmzeilen einen Kreis in kobaltblau und malen dann das Innere des Kreises violett aus:

```
SCREEN 1
CIRCLE (160, 100), 50, 1
PAINT (160, 100), 2, 1
```

Für das Ausmalen von Figuren gelten drei Regeln:

1. Die in der **PAINT**-Anweisung angegebenen Koordinaten müssen sich auf einen Punkt innerhalb der Figur beziehen.

Zum Beispiel würde jede der folgenden Anweisungen dieselbe Auswirkung wie die in den zwei vorhergehenden Beispielen gezeigten **PAINT**-Anweisungen haben, da alle Koordinaten Punkte innerhalb des Kreises kennzeichnen:

```
PAINT (150, 90), 2, 1
PAINT (170, 110), 2, 1
PAINT (180, 80), 2, 1
```

Da `(5, 5)` einen Punkt außerhalb des Kreises kennzeichnet, würde dagegen die nächste Anweisung den gesamten Bildschirm mit Ausnahme des Kreisinneren, das die aktuelle Hintergrundfarbe behält, ausmalen:

```
PAINT (5, 5), 2, 1
```

Wenn die Koordinaten einer **PAINT**-Anweisung einen Punkt direkt auf dem Rand der Figur angeben, findet der Ausmalvorgang nicht statt:

```
LINE (50, 50)-(150, 150), , B  ' Zeichne ein Rechteck.
PAINT (50, 100)        ' Der Punkt mit den Koordinaten
                       ' (50, 100) liegt auf dem oberen
                       ' Rand des Rechtecks, daher wird
                       ' nicht ausgemalt.
```

2. Die Figur muß ganz geschlossen sein; andernfalls "läuft" die Malfarbe "aus", wodurch der gesamte Bildschirm oder das gesamte Darstellungsfeld (oder jede größere Figur, die die erste ganz umschließt) ausgefüllt wird.

   In dem folgenden Programm zeichnet die Anweisung **CIRCLE**, zum Beispiel, eine nicht ganz geschlossene Ellipse (auf der rechten Seite befindet sich eine kleine Öffnung); die Anweisung **LINE** umschließt dann die Teilellipse mit einem Rechteck. Obwohl das Ausmalen im Inneren der Ellipse beginnt, fließt die Malfarbe durch die Öffnung und füllt das gesamte Rechteck aus.

```
SCREEN 2
CONST PI = 3.141592653589#
CIRCLE (300, 100), 80, , 0, 1.9 * PI, 3
LINE (200, 10)-(400, 190), , B
PAINT (300, 100)
```

3. Wenn Sie ein Objekt mit einer anderen Farbe als die der Umrandung des Objektes ausmalen, ist die Option *Rand* zu verwenden, um **PAINT** mitzuteilen, wo das Ausmalen beendet werden soll.

   Zum Beispiel zeichnet das folgende Programm ein grün (Attribut 1 in Palette 0) umrandetes Dreieck und versucht anschließend, das Innere des Dreiecks rot (Attribut 2) auszumalen. Da die **PAINT**-Anweisung jedoch nicht anzeigt, wo das Ausmalen beendet werden soll, malt diese den gesamten Bildschirm rot aus.

```
SCREEN 1
COLOR , 0

LINE (10, 25)-(310, 25), 1
LINE -(160, 175), 1
LINE -(10, 25), 1

PAINT (160, 100), 2
```

   Folgende Änderung der **PAINT**-Anweisung (für das Innere rot wählen und beenden, wenn ein grünfarbiger Rand erreicht wird) erzielt den gewünschten Effekt:

```
PAINT (160, 100), 2, 1
```

   Es ist zu beachten, daß Sie in der **PAINT**-Anweisung keine Randfarbe anzugeben brauchen, wenn die Farbe zum Ausmalen der Randfarbe entspricht.

```
LINE (10, 25)-(310, 25), 1
LINE -(160, 175), 1
LINE -(10, 25), 1
PAINT (160, 100), 1
```

5.8.2 Zeichnen mit Mustern: Ausfüllen mit Mustern

Anhand der **PAINT**-Anweisung läßt sich eine geschlossene Figur mit einem Muster ausfüllen: Dieser Vorgang wird als "Ausfüllen mit Mustern" bezeichnet. Ein Muster ähnelt einer "Kachel", die den Baustein einer bemusterten Fläche darstellt und das Verfahren gleicht dem Kachelnlegen auf dem Fußboden. Wird Ausfüllen mit Mustern verwendet, ist das Argument *Innen* in der Syntax von **PAINT** ein Zeichenkettenausdruck, nicht eine Zahl. Da *Innen* jeder Zeichenkettenausdruck sein kann, besteht eine bequeme Möglichkeit zur Definition von "Kachelmustern" für *Innen* in der folgenden Form:

CHR$**(*Arg1*) + **CHR$(*Arg2*) + **CHR$**(*Arg3*) + ... + **CHR$**(*Argn*)

Hier sind *Arg1*, *Arg2* und so weiter 8-Bit-Ganzzahlen. Eine Erklärung zur Bildung dieser 8-Bit-Ganzzahlen, finden Sie in den Abschnitten 5.8.2.2 bis 5.8.2.4.

5.8.2.1 Größe eines Kachelmusters in unterschiedlichen Bildschirmmodi

Jede Kachel eines Musters ist aus einem rechteckigen Raster von Bildpunkten zusammengesetzt. Dieser Kachelraster kann in allen Bildschirmmodi bis zu 64 Zeilen umfassen. Die Anzahl an Bildpunkten in jeder Zeile hängt jedoch vom Bildschirmmodus ab.

Die Länge jeder Kachelzeile variiert aus folgenden Gründen je nach Bildschirmmodus: Obwohl die Anzahl an Bits in jeder Zeile auf acht festgelegt ist (die Länge einer Ganzzahl), verkleinert sich die Anzahl an Bildpunkten, die diese acht Bits darstellen können, in gleichem Maße, wie sich die Anzahl der Farbattribute in einem gegebenen Bildschirmmodus vergrößert. Zum Beispiel ist im Bildschirmmodus 2, der nur 1 Farbattribut hat, die Anzahl der Bits pro Bildpunkt gleich 1; im Bildschirmmodus 1, der 4 verschiedene Attribute hat, ist die Anzahl der Bits pro Bildpunkt gleich 2; im EGA-Bildschirmmodus 7, der 16 Attribute hat, ist die Anzahl der Bits pro Bildpunkt gleich 4. Folgende Formel ermöglicht die Berechnung der Bits pro Bildpunkt im jeweiligen Bildschirmmodus:

Bits pro Bildpunkt=LOG_2(*AnzAttribute*)

Hierbei ist *AnzAttribute* die Anzahl an Farbattributen in diesem Bildschirmmodus. (Der angeschlossene QB-Ratgeber erteilt diese Information.)

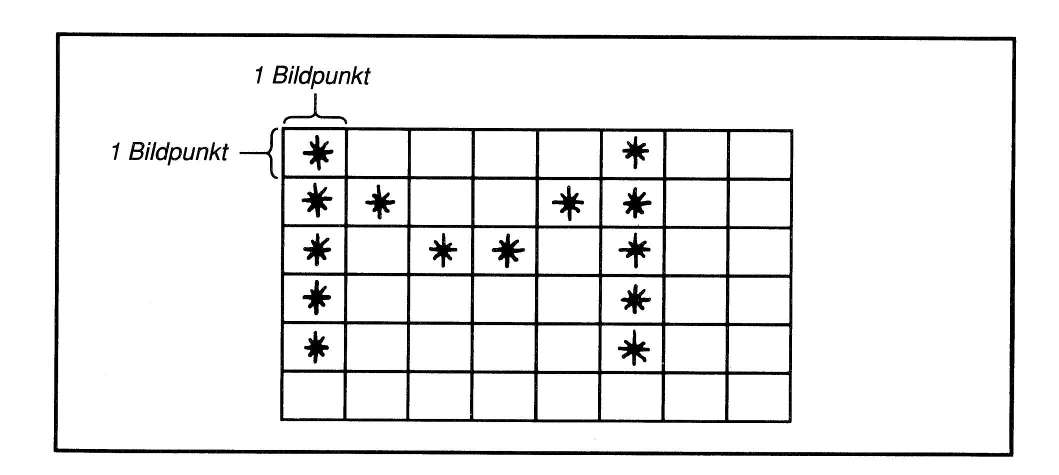
Daher beträgt die Länge einer Kachelzeile im Bildschirmmodus 2 acht Bildpunkte (acht Bits dividiert durch ein Bit pro Bildpunkt), aber nur vier Bildpunkte im Bildschirmmodus 1 (acht Bits dividiert durch zwei Bits pro Bildpunkt).

Die nächsten drei Abschnitte zeigen den schrittweisen Ablauf zum Erzeugen eines Kachelmusters. Abschnitt 5.8.2.2 erklärt das Erzeugen eines monochromen Musters im Bildschirmmodus 2. Der nächste Abschnitt 5.8.2.3 erläutert das Erstellen eines vielfarbigen Musters im Bildschirmmodus 1. Wenn Sie eine EGA-Karte haben, können Sie in Abschnitt 5.8.2.4 nachlesen, wie im Bildschirmmodus 8 ein vielfarbiges Muster erzeugt wird.

5.8.2.2 Erzeugen eines einfarbigen Musters im Bildschirmmodus 2

Nachstehende Arbeitsschritte zeigen, wie ein M-förmiges Kachelmuster zu definieren und anzuwenden ist:

1. Zeichnen Sie das Muster einer Kachel in einem 8-spaltigen Raster, der die benötigte Zeilenanzahl (höchstens 64) enthält. In diesem Beispiel hat die Kachel 6 Zeilen; ein Stern (*) in einem Rechteck bedeutet, daß der Bildpunkt eingeschaltet ist:



2. Übersetzen Sie nun jede Bildpunktzeile in eine 8-Bitzahl, wobei 1 bedeutet, daß der Bildpunkt eingeschaltet und 0, daß er ausgeschaltet ist:

1	0	0	0	0	1	0	0
1	1	0	0	1	1	0	0
1	0	1	1	0	1	0	0
1	0	0	0	0	1	0	0
1	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0

3. Wandeln Sie die in Schritt 2 gegebenen binären Zahlen in hexadezimale Ganzzahlen um:

```
10000100 = &H84
11001100 = &HCC
10110100 = &HB4
10000100 = &H84
10000100 = &H84
00000000 = &H00
```

Diese Ganzzahlen müssen nicht hexadezimal sein; sie können auch dezimal oder oktal sein. Die Umwandlung von binär in hexadezimal scheint jedoch leichter zu sein. Um von binär in hexadezimal umzuwandeln, ist die Binärzahl von rechts nach links zu lesen. Anschließend wird jede vierstellige Ziffergruppe in die hexadezimale Entsprechung wie folgt umgewandelt:

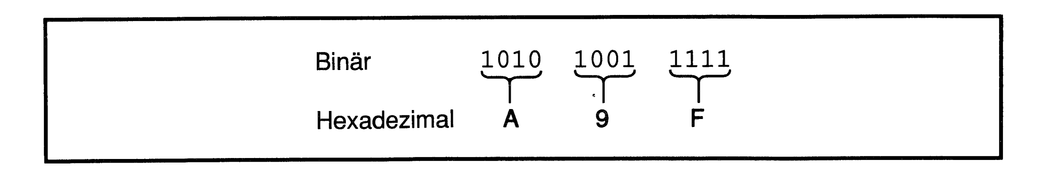


Tabelle 5.3 führt 4-Bit-Binärsequenzen und deren hexadezimale Entsprechungen auf.

4. Erzeugen Sie eine Zeichenkette durch Verketten der Zeichen mit den ASCII-Werten aus Schritt 3 (erstellen Sie diese Zeichen anhand der Funktion **CHR$**):

```
Kachel$ = CHR$(&H84) + CHR$(&HCC) + CHR$(&HB4)
Kachel$ = Kachel$ + CHR$(&H84) + CHR$(&H84) + CHR$(&H00)
```

5. Zeichnen Sie eine Figur und malen Sie deren Inneres mit Hilfe der Anweisung **PAINT** und des Zeichenkettenargumentes aus Schritt 4 aus:

```
PAINT (X, Y), Kachel$
```

Tabelle 5.3 Umwandlung von binär in hexadezimal

Binärzahl	***Hexadezimalzahl***
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

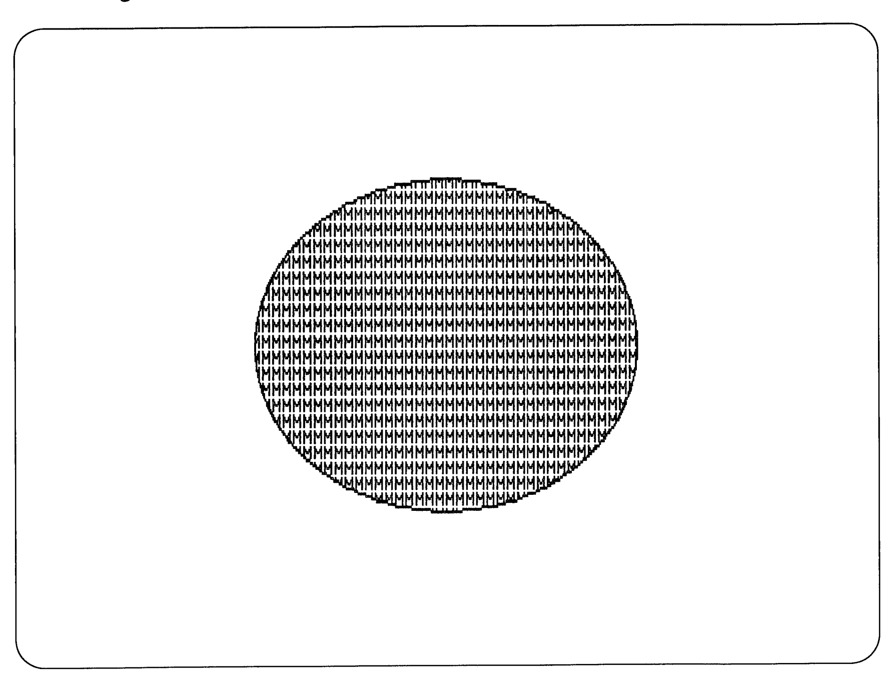
Beispiel

Das folgende Beispiel zeichnet einen Kreis und malt anschließend das Kreisinnere mit dem in den vorhergehenden Schritten erzeugten Muster aus.

```
SCREEN 2
CLS
Kachel$ = CHR$(&H84) + CHR$(&HCC) + CHR$(&HB4)
Kachel$ = Kachel$ + CHR$(&H84) + CHR$(&H84) + CHR$(&H00)
CIRCLE STEP (0, 0), 150
PAINT STEP (0, 0), Kachel$
```

Ausgabe

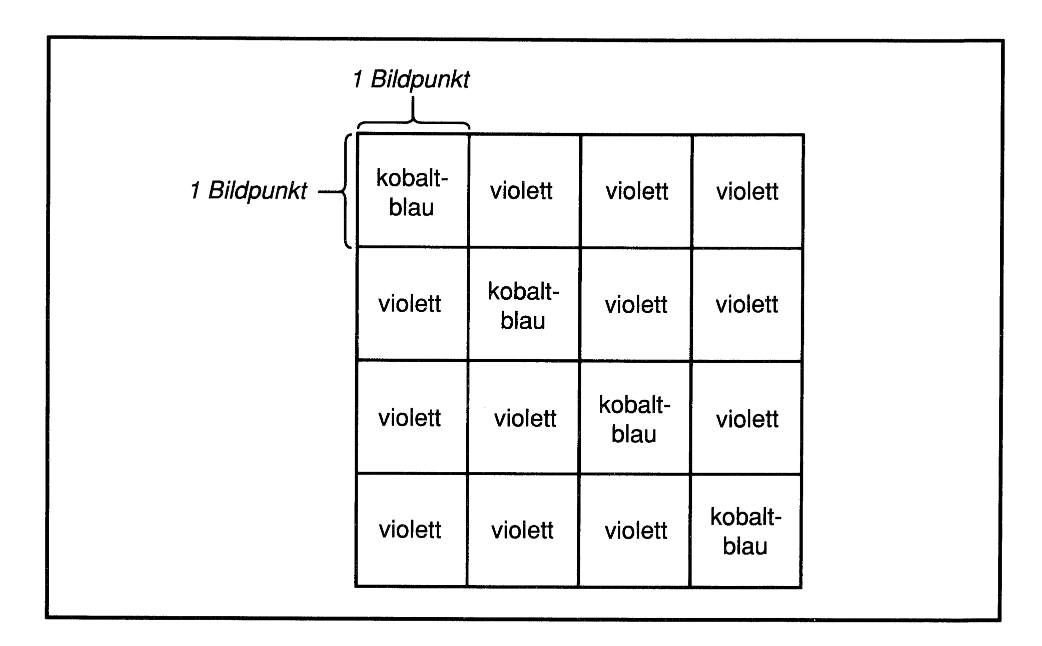
Abbildung 5.5 Gemusterter Kreis



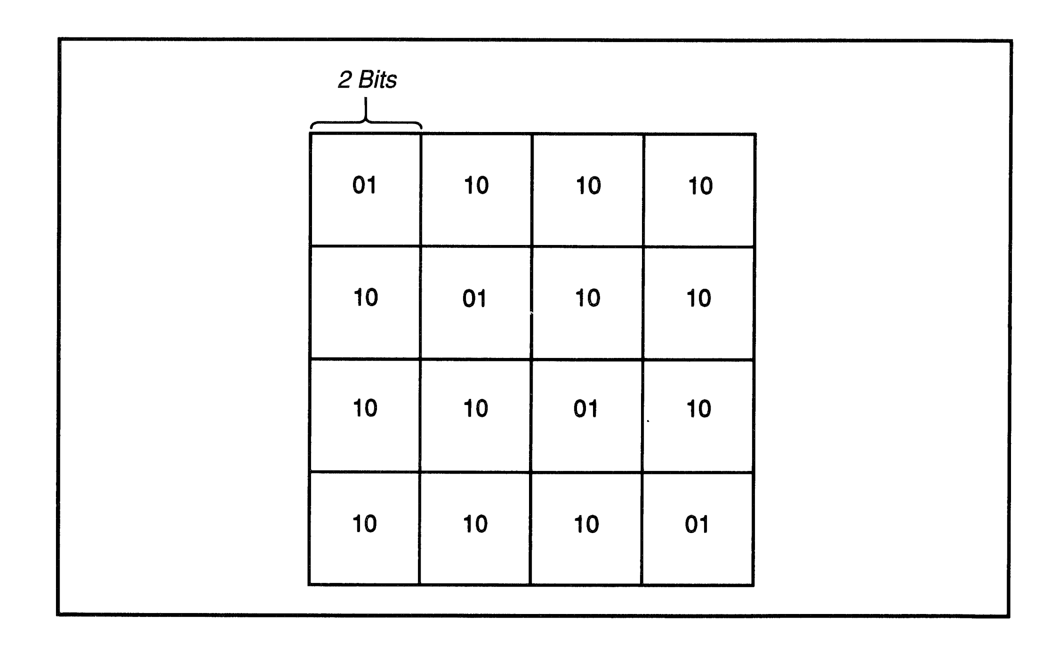
5.8.2.3 Erzeugen eines vielfarbigen Musters im Bildschirmmodus 1

Die folgenden Schritte zeigen, wie Sie ein vielfarbiges Muster erzeugen, das aus alternierenden kobaltblauen bzw. violetten diagonalen Streifen besteht (oder grün bzw. rot mit Palette 0):

1. Zeichnen Sie das Kachelmuster in einem 4-spaltigen Raster (4 Spalten, da jede Bildpunktzeile in einer 8-Bit-Ganzzahl gespeichert ist und jeder Bildpunkt im Bildschirmmodus 1 zwei Bits erfordert) mit der notwendigen Zeilenanzahl (bis zu 64). In diesem Beispiel hat die Kachel 4 Zeilen, wie im nächsten Diagramm gezeigt.



2. Wandeln Sie die Farben wie unten gezeigt in deren zugehörige Farbnummern in binärer Schreibweise um (vergewissern Sie sich, daß Sie 2-Bitwerte verwenden, also 1 = binär 01 und 2 = binär 10):



3. Wandeln Sie die Binärzahlen aus Schritt 2 in hexadezimale Ganzzahlen um:

```
01101010 = &H6A
10011010 = &H9A
10100110 = &HA6
10101001 = &HA9
```

4. Erzeugen Sie eine Zeichenkette durch Verketten der Zeichen mit den ASCII-Werten aus Schritt 3 (erstellen Sie diese Zeichen mit Hilfe der Funktion **CHR$**):

```
Kachel$ = CHR$(&H6A) + CHR$(&H9A) + CHR$(&HA6)+ _
          CHR$(&HA9)
```

5. Zeichnen Sie eine Abbildung und malen Sie deren Inneres anhand der Anweisung **PAINT** und des Zeichenkettenargumentes aus Schritt 4 aus:

```
PAINT (X, Y), Kachel$
```

Das folgende Programm zeichnet ein Dreieck und malt anschließend dessen Inneres mit dem in den vorhergehenden Schritten erzeugten Muster aus:

```
SCREEN 1
' Definiere ein Muster:
Kachel$ = CHR$(&H6A) + CHR$(&H9A) + CHR$(&HA6) + CHR$(&HA9)
' Zeichne ein Dreieck in weiß (Farbe 3):
LINE (10, 25)-(310, 25)
LINE -(160, 175)
LINE -(10, 25)
' Male das Innere des Dreiecks mit dem Muster aus:
PAINT (160, 100), Kachel$
```

Wenn die auszumalende Figur mit einer Farbe umrandet ist, die auch in dem Muster vorkommt, muß in **PAINT** das Argument *Rand*, wie nachfolgend gezeigt, angegeben werden; andernfalls geht das Muster über die Ränder der Figur hinaus.

```
SCREEN 1
' Definiere ein Muster:
Kachel$ = CHR$(&H6A) + CHR$(&H9A) + CHR$(&HA6) + CHR$(&HA9)
' Zeichne ein Dreieck in violett (Farbe 2):
LINE (10, 25)-(310, 25), 2
LINE -(160, 175), 2
LINE -(10, 25), 2
' Male das Innere des Dreiecks mit dem Muster aus und füge
' das Randargument (, 2) hinzu, um PAINT mitzuteilen, wo das
' Ausmalen beendet werden soll:
PAINT (160, 100), Kachel$, 2
```

Es kann vorkommen, daß Sie eine bereits mit einer Farbe oder einem Muster ausgemalte Abbildung oder einige ihrer Teile neu ausmalen möchten. Wenn das neue Muster zwei oder mehrere angrenzende Zeilen enthält, die identisch mit dem aktuellen Hintergrund der Abbildung sind, werden Sie feststellen, daß das Ausmalen mit Mustern nicht funktioniert. Stattdessen breitet sich das Muster aus, umgeben von Bildpunkten, die mit zwei oder mehreren seiner Zeilen identisch sind, und das Ausmalen wird beendet.

Dieses Problem kann durch Einsetzen des Arguments *Hintergrund* in der Anweisung **PAINT** vermindert werden, wenn sich nicht mehr als zwei angrenzende Zeilen in dem neuen Muster befinden, die das gleiche Muster wie der alte Hintergrund aufweisen. **PAINT** mit *Hintergrund* hat folgende Syntax:

PAINT [STEP] *(x,y)* **[,[***Innen***][,[***Rand***][,** *Hintergrund***]]]**

Das Argument *Hintergrund* ist ein Zeichen der Form **CHR$(***n***)**, das die Zeilen im Kachelmuster angibt, die dieselben Muster aufweisen wie die des aktuellen Abbildungshintergrundes. Im wesentlichen teilt *Hintergrund* der Anweisung **PAINT** mit, diese Zeilen beim erneuten Ausmalen der Abbildung zu überspringen. Das nächste Beispiel verdeutlicht diesen Vorgang.

```
SCREEN 1

' Definiere ein Muster (jeweils zwei Zeilen sind kobaltblau,
' violett, weiß):
Kachel$ = CHR$(&H55) + CHR$(&H55) + CHR$(&HAA)
Kachel$ = Kachel$ + CHR$(&HAA) + CHR$(&HFF) + CHR$(&HFF)

' Zeichne ein Dreieck in weiß (Farbe Nummer 3):
LINE (10, 25)-(310, 25)
LINE -(160, 175)
LINE -(10, 25)

' Male das Innere in violett aus:
PAINT (160, 100), 2, 3

' Warte auf eine Tastenbetätigung:
Pause$ = INPUT$(1)

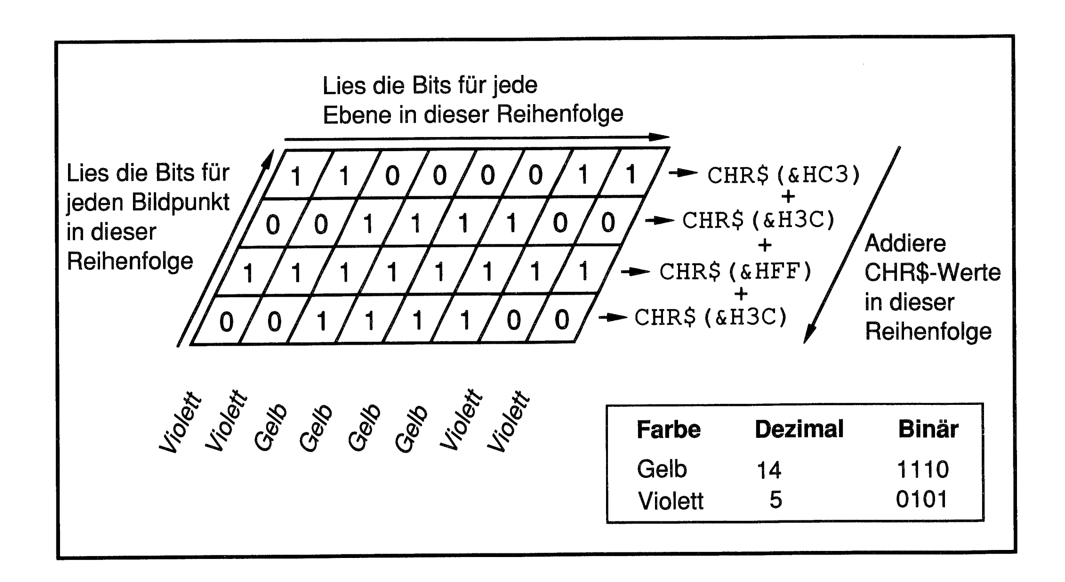
' Da der Hintergrund bereits violett ist, teilt CHR$(&HAA)
' PAINT mit, die violetten Zeilen des Kachelmusters zu
' überspringen:
PAINT (160, 100), Kachel$, , CHR$(&HAA)
```

5.8.2.4 Erzeugen eines vielfarbigen Musters im Bildschirmmodus 8

In den Bildschirmmodi EGA bzw. VGA ist mehr als eine 8-Bit-Ganzzahl erforderlich, um die Zeile einer Musterkachel zu definieren. In diesen Bildschirmmodi besteht eine Zeile aus mehreren Ebenen von 8-Bit-Ganzzahlen. Der Grund dafür ist, daß ein Bildpunkt dreidimensional in einem Stapel von "Bitebenen", nicht sequentiell in einer einzigen Ebene dargestellt wird, wie es in den Bildschirmmodi 1 und 2 der Fall ist. Zum Beispiel hat der Bildschirmmodus 8 vier dieser Bitebenen. Daher befinden sich die jeweiligen vier Bits pro Bildpunkt in diesem Bildschirmmodus auf unterschiedlichen Ebenen.

Nachstehende Schritte veranschaulichen den Erstellungsablauf eines vielfarbigen Musters, das aus abwechselnd gelben und violetten Zeilen besteht. Beachten Sie, wie jede Zeile der Musterkachel durch vier parallele Bytes dargestellt wird:

1. Definieren Sie eine Zeile von Bildpunkten in der Musterkachel. Jeder Bildpunkt in der Zeile beansprucht vier Bits und jedes Bit befindet sich auf einer anderen Ebene, wie in nachfolgender Abbildung dargestellt:

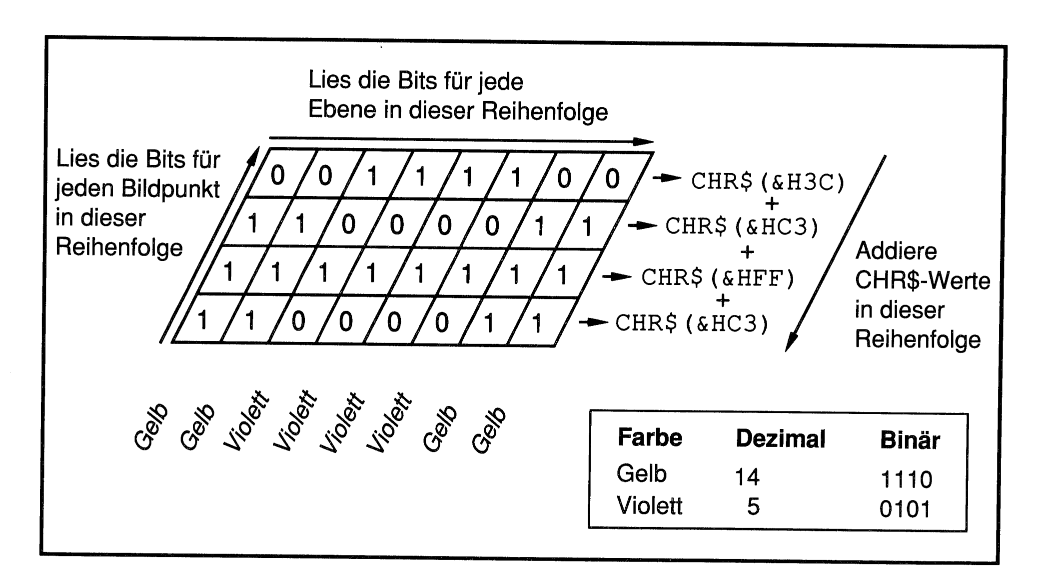


Farbe	Dezimal	Binär
Gelb	14	1110
Violett	5	0101

Addieren Sie die **CHR$**-Werte aller vier Bitebenen, um einen Kachelbyte zu erhalten. Diese Zeile wird in der Musterkachel wiederholt, daraus folgt:

Zeile$(1) = Zeile$(2)=
CHR$**(&HC3) + **CHR$(&H3C) + **CHR$**(&HFF) + **CHR$**(&H3C)

2. Definieren Sie eine weitere Zeile von Bildpunkten in der Musterkachel, wie folgt:



Farbe	Dezimal	Binär
Gelb	14	1110
Violett	5	0101

Diese Zeile wird ebenfalls in der Musterkachel wiederholt und daraus folgt:

Zeile$(3) = Zeile$(4)=
CHR$**(&H3C) + **CHR$(&HC3) + **CHR$**(&HFF) + **CHR$**(&HC3)

Beispiel

Das folgende Beispiel zeichnet ein Rechteck und malt anschließend dessen Inneres mit dem in den vorhergehenden Schritten erzeugten Muster aus:

```
SCREEN 8
DIM Zeile$(1 TO 4)
' Zwei Zeilen mit abwechselnd violett und gelb:
Zeile$(1) = CHR$(&HC3) + CHR$(&H3C) + CHR$(&HFF) + _
            CHR$(&H3C)
Zeile$(2) = Zeile$(1)
' Drehe das Muster um (zwei Zeilen abwechselnd gelb und
' violett):
Zeile$(3) = CHR$(&H3C) + CHR$(&HC3) + CHR$(&HFF) + _
            CHR$(&HC3)
Zeile$(4) = Zeile$(3)
```

```
' Erzeuge eine Musterkachel mit den oben definierten Zeilen:
FOR I% = 1 TO 4
   Kachel$ = Kachel$ + Zeile$(I%)
NEXT I%
' Zeichne ein Rechteck und fülle es mit dem Muster aus:
LINE (50, 50)-(570, 150), , B
PAINT (320, 100), Kachel$
```

5.9 DRAW: Eine graphische Makro-Sprache

Die Anweisung **DRAW** bildet selbst eine "Miniatursprache". Sie zeichnet und malt Bilder auf den Bildschirm, wobei sie als "Makros" bezeichnete Befehlssätze verwendet, die aus einem oder zwei Buchstaben bestehen und in einen Zeichenkettenausdruck eingefügt sind.

DRAW bietet folgende Vorteile gegenüber den anderen bisher erläuterten Graphikanweisungen:

- Das Makro-Zeichenkettenargument in **DRAW** ist zusammengefaßt: Eine einzige kurze Zeichenkette kann dieselbe Ausgabe erzeugen wie mehrere **LINE**-Anweisungen.
- Mit **DRAW** erzeugte Bilder können leicht maßstabgetreu verändert, bzw. verkleinert oder vergrößert werden, indem das Makro S in der Makro-Zeichenkette verwendet wird.
- Mit **DRAW** erzeugte Bilder können um eine beliebige Gradzahl gedreht werden, indem in der Makro-Zeichenkette das Makro **TA** verwendet wird.

Nähere Informationen sind dem QB-Ratgeber zu entnehmen.

Beispiel

Das folgende Programm ist eine kurze Einführung in die Bewegungsmakros **U**, **D**, **L**, **R**, **E**, **F**, **G** und **H**, den Makro **B** "zeichne/zeichne nicht" und den Farbmakro **C**. Dieses Programm zeichnet horizontale, vertikale und diagonale Geraden in unterschiedlichen Farben je nach den auf dem Zehnerblock betätigten RICHTUNGSTASTEN (↑, →, ↓, BILD ↑, BILD ↓ etc.).

Dieses Programm befindet sich in der Datei mit dem Namen *plotter.bas* auf den QuickBASIC-Originaldisketten.

```
' Werte für Tasten auf dem Zehnerblock und die
' Leertaste:
CONST OBEN = 72, UNTEN = 80, LI = 75, RE = 77
CONST OBENLI = 71, OBENRE = 73, UNTLI = 79, UNTRE = 81
CONST LEERZEICH = " "

' Null$ ist das erste Zeichen des Zweizeichen-Wertes,
' den INKEY$ für Richtungstasten wie OBEN und UNTEN
' zurückgibt:
Null$ = CHR$(0)

' Zeichne$ = "" bedeutet zeichne Geraden;
' Zeichne$ = "B" bedeutet bewege den grafischen
' Cursor, aber zeichne keine Geraden:
Zeichne$ = ""

PRINT "Benutzen Sie die Richtungstasten, um Geraden _
 zu zeichnen."
PRINT "Betätigen Sie die Leertaste, um das Zeichnen _
 von Geraden ein- bzw. auszuschalten."
PRINT "Betätigen Sie <EINGABETASTE>, um zu beginnen."
PRINT "Betätigen Sie E zum Beenden des Programmes."
DO : LOOP WHILE INKEY$ = ""

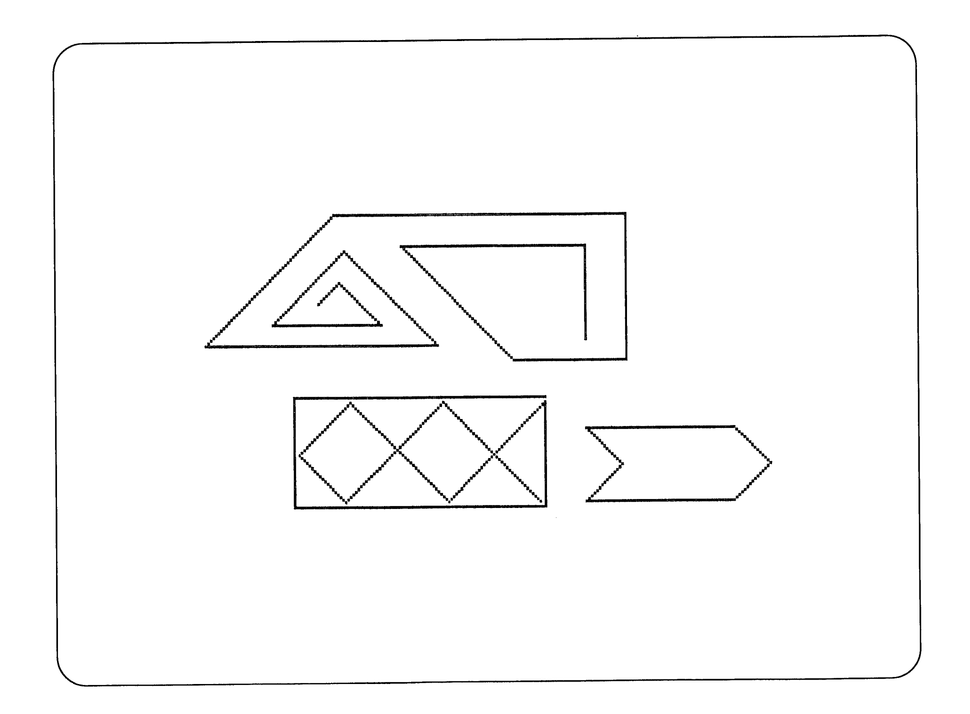
SCREEN 1

DO
   SELECT CASE TastenWert$
      CASE Null$ + CHR$(OBEN)
         DRAW Zeichne$ + "C1 U2"
      CASE Null$ + CHR$(UNTEN)
         DRAW Zeichne$ + "C1 D2"
      CASE Null$ + CHR$(LI)
         DRAW Zeichne$ + "C2 L2"
      CASE Null$ + CHR$(RE)
         DRAW Zeichne$ + "C2 R2"
      CASE Null$ + CHR$(OBENLI)
         DRAW Zeichne$ + "C3 H2"
      CASE Null$ + CHR$(OBENRE)
         DRAW Zeichne$ + "C3 E2"
      CASE Null$ + CHR$(UNTLI)
         DRAW Zeichne$ + "C3 G2"
      CASE Null$ + CHR$(UNTRE)
         DRAW Zeichne$ + "C3 F2"
```

```
      CASE LEERZEICH
         IF Zeichne$ = "" THEN
            Zeichne$ = "B "
         ELSE
            Zeichne$ = ""
         END IF
      CASE ELSE
         ' Der Benutzer betätigte eine andere Taste
         ' als eine der Richtungstasten, der Leertaste
         ' oder "E", also tue nichts.
   END SELECT
   TastenWert$ = INKEY$
LOOP UNTIL TastenWert$ = "E"
SCREEN 0, 0       ' Stelle den 80-Spalten Textmodus
                  ' wieder her und beende.
WIDTH 80
END
```

Ausgabe

Untenstehend wird ein durch dieses Programm erzeugter Musterentwurf aufgeführt.



5.10 Grundlegende Techniken der Animation

Anhand der in den vorherigen Abschnitten behandelten Graphikanweisungen können Sie einfache Animation mit Objekten auf dem Bildschirm ausführen. Zum Beispiel können Sie zunächst mit **CIRCLE** einen Kreis malen, diesen anschließend mit der Hintergrundfarbe neu zeichnen, um ihn zu löschen, und zum Schluß den Mittelpunkt des Kreises erneut berechnen, um diesen an einer anderen Stelle zu zeichnen.

Diese Technik reicht für einfache Figuren aus, jedoch werden deren Nachteile beim Aufbau komplexerer Bilder offensichtlich. Trotz der Geschwindigkeit der Graphikanweisungen sind die Verzögerungen dennoch bemerkbar. Darüberhinaus ist es mit dieser Methode nicht möglich, den Hintergrund beizubehalten: Durch Entfernen des Objekts wird alles entfernt, was sich bereits auf dem Bildschirm befand.

Zwei Anweisungen, **GET** und **PUT**, ermöglichen die Durchführung schneller Animation. Sie können ein Bild mit Hilfe der Ausgaben von Anweisungen wie **PSET**, **LINE**, **CIRCLE** oder **PAINT** erzeugen und anschließend einen "Schnappschuß" des Bildes mit **GET** machen, um das Bild in den Speicher zu kopieren. Nachher können Sie anhand von **PUT** das mit **GET** gespeicherte Bild an beliebiger Stelle auf dem Bildschirm bzw. Darstellungsfeld reproduzieren.

5.10.1 Speichern der Bilder mit GET

Nach Erzeugung des Originalbildes auf dem Bildschirm, müssen die *x*- und *y*-Koordinaten eines Rechteckes genügend groß zur Aufnahme des gesamten Bildes berechnet werden. Dann kann mit Hilfe von **GET**, das gesamte Rechteck in den Speicher kopiert werden. Die Syntax der graphischen Anweisung **GET** ist

GET [STEP] (*x1*,*y1*) - **[STEP]** (*x2*,*y2*), *Datenfeldname*

wobei (*x1*,*y1*) und (*x2*,*y2*) die Koordinaten der oberen linken bzw. unteren rechten Ecke des Rechtecks angeben. *Datenfeldname* ist ein beliebiges numerisches Datenfeld. Die in einer **DIM**-Anweisung für *Datenfeldname* angegebene Größe hängt von folgenden drei Faktoren ab:

1. der Höhe und Breite des Rechtecks, das das Bild einschließt.
2. dem für die graphische Ausgabe gewählten Bildschirmmodus.
3. dem Typ des Datenfeldes (Ganzzahl, lange Ganzzahl, einfache Genauigkeit oder doppelte Genauigkeit).

Hinweis Obwohl das zum Speichern von Bildern verwendete Datenfeld jeden numerischen Typ haben kann, ist es sehr empfehlenswert, nur ganzzahlige Datenfelder zu verwenden. Alle graphischen Muster, die auf dem Bildschirm möglich sind, können mit Ganzzahlen dargestellt werden. Dies ist jedoch nicht der Fall bei reellen Zahlen einfacher oder doppelter Genauigkeit: Einige graphische Muster sind keine gültigen reellen Zahlen und deren Speicherung in einem Datenfeld reeller Zahlen könnte zu unvorhergesehenen Ergebnissen führen.

Die Formel zur Berechnung der Größe von *Datenfeldname* in Bytes ist

*Größe in Bytes = 4 + Höhe * Ebenen * **INT**((Breite * Bits pro Bildpunkt/Ebenen + 7)/8)*

wobei *Höhe* und *Breite* die Dimensionen in Bildpunktanzahl des zu erhaltenden Rechteckes darstellen; der Wert der *Bits pro Bildpunkt* hängt von der Anzahl der in dem gegebenen Bildschirmmodus verfügbaren Farben ab. Je höher die Anzahl der Farben, desto höher ist die Anzahl der zur Definition jedes Bildpunktes benötigten Bits. Im Bildschirmmodus 1 definieren zwei Bits einen Bildpunkt, während im Bildschirmmodus 2 ein Bit ausreicht. (Die allgemeine Formel für Bits pro Bildpunkt finden Sie in Abschnitt 5.8.2.1., "Größe eines Kachelmusters in unterschiedlichen Bildschirmmodi".) Die folgende Liste zeigt die Werte für Ebenen im jeweiligen Bildschirmmodus:

Bildschirmmodi	***Anzahl von Bit-Ebenen***
1, 2, 11 und 13	1
9 (64K Videospeicher) und 10	2
7, 8, 9 (mehr als 64K Videospeicher) und 12	4

Um die Anzahl von Elementen, die das Datenfeld haben sollte, zu erhalten, dividieren Sie *Größe in Bytes* durch die Byteanzahl eines Elementes im Datenfeld. Demnach ist jetzt der Typ des Datenfeldes von Bedeutung. Wenn es sich um ein ganzzahliges Datenfeld handelt, belegt jedes Element zwei Bytes im Speicher (die Größe einer Ganzzahl), so daß *Größe in Bytes* durch zwei dividiert werden muß, um die tatsächliche Größe des Datenfeldes zu erhalten. Wenn das Datenfeld den Typ lange Ganzzahl hat, muß *Größe in Bytes* durch vier dividiert werden (da eine lange Ganzzahl vier Bytes im Speicher beansprucht) usw. Hat das Datenfeld den Typ einfache Genauigkeit, dividieren Sie durch vier; hat es den Typ doppelte Genauigkeit, dividieren Sie durch acht.

Folgende Schritte zeigen die Berechnung der Größe eines ganzzahligen Datenfeldes, das ein Rechteck mit den Koordinaten `(10,40)` für die obere linke Ecke und `(90,80)` für die untere rechte Ecke im Bildschirmmodus 1 aufnehmen kann:

1. Berechnen Sie die Höhe und Breite des Rechtecks:

RechtHoehe = **ABS**(*y2 - y1*) + 1 = 80 - 40 + 1 = 41
RechtBreite = **ABS**(*x2 - x1*) + 1 = 90 - 10 + 1 = 81

Vergessen Sie nicht, nach der Subtraktion *y1* von *y2* bzw. *x1* von *x2* eine Eins hinzuzuaddieren. Wenn z. B. *x1* = 10 und *x2* = 20 ist, dann ist die Breite des Rechtecks 20 - 10 + 1 oder 11.

2. Berechnen Sie die Größe des Ganzzahldatenfeldes in Bytes:

   ByteGroesse = 4 + RechtHoehe * **INT**((RechtBreite * BitsProPixel + 7)/8)
   = 4 + 41 * **INT**((81 * 2 + 7) / 8)
   = 4 + 41 * **INT**(169/8)
   = 4 + 41 * 21
   = 865

3. Dividieren Sie die Größe in Bytes durch die Byteanzahl pro Element (zwei für Ganzzahlen) und runden Sie das Ergebnis auf die nächste Ganzzahl auf:

   865/2 = 433

Wenn der Name des Datenfeldes `Bild` ist, gewährleistet die folgende **DIM**-Anweisung, daß `Bild` genügend groß ist, die Information der Bildpunkte im Rechteck zu kopieren:

```
DIM Bild (1 TO 433) AS INTEGER
```

Hinweis Obwohl die **GET**-Anweisung nach einer **WINDOW**-Anweisung logische Koordinaten verwendet, müssen zur Berechnung der Größe des in **GET** verwendeten Datenfeldes physikalische Koordinaten benutzt werden. (Weitere Informationen zu **WINDOW** und zur Umwandlung logischer in physikalische Koordinaten finden Sie im obigen Abschnitt 5.6, "Neu Definieren von Koordinaten eines Darstellungsfeldes anhand von **WINDOW**".)

Die oben erwähnten Schritte geben die für das Datenfeld erforderliche Mindestgröße an, jedoch ist jegliche umfangreichere Größe ebenfalls zweckmäßig. Zum Beispiel gilt auch die Anweisung:

```
DIM Bild (1 TO 500) AS INTEGER
```

Beispiel

Das folgende Programm zeichnet eine Ellipse und malt deren Inneres aus. Eine **GET**-Anweisung kopiert den rechteckigen Bereich, der die Ellipse enthält, in den Speicher. (Abschnitt 5.10.2, "Bewegung der Bilder mit PUT" zeigt, wie die Ellipse anhand der Anweisung **PUT** an einer anderen Stelle reproduziert werden kann.)

```
SCREEN 1
' Dimensioniere ein ganzzahliges Datenfeld, das genügend
' groß ist, das Rechteck aufzunehmen:
DIM Bild (1 TO 433) AS INTEGER
' Zeichne innerhalb des Rechteckes eine Ellipse, verwende
' violett für die Umrandung und male das Innere weiß aus:
CIRCLE (50, 60), 40, 2, , , .5
PAINT (50, 60), 3, 2
' Speichere das Abbild des Rechtecks in dem Datenfeld:
GET (10, 40)-(90, 80), Bild
```

5.10.2 Bewegung der Bilder mit PUT

Die **GET**-Anweisung erlaubt die Erstellung der Momentaufnahme eines Bildes, während die Anweisung **PUT** das Bild an jeder gewünschten Stelle auf dem Bildschirm einfügt. Eine Anweisung der Form

PUT (*x,y*) ,*Datenfeldname*[, *Aktionswort*]

kopiert das in *Datenfeldname* gespeicherte rechteckige Bild zurück auf den Bildschirm und plaziert es mit der oberen linken Ecke an dem Punkt mit den Koordinaten (*x,y*). Es ist zu beachten, daß **PUT** nur ein Koordinatenpaar aufweist.

Wenn in dem Programm vor **PUT** eine **WINDOW**-Anweisung erscheint, beziehen sich die Koordinaten *x* und *y* auf die untere linke Ecke des Rechtecks. **WINDOW SCREEN** hat jedoch nicht diese Auswirkung; d. h., nach **WINDOW SCREEN** beziehen sich *x* und *y* nach wie vor auf die obere linke Ecke des Rechtecks.

Zum Beispiel bewirkt das Hinzufügen der nächsten Zeile zu dem vorhergehenden Beispiel in Abschnitt 5.10.1 viel schneller das Erscheinen einer exakten Kopie der ausgemalten Ellipse auf der rechten Seite des Bildschirms als das neu Zeichnen und Ausmalen derselben Abbildung mit **CIRCLE** und **PAINT**:

```
PUT (200, 40), Bild
```

Vergewissern Sie sich, daß die angegebenen Koordinaten nicht irgendeinen Teil des Abbildes außerhalb des Bildschirms oder des aktiven Darstellungsfeldes positionieren, wie in dem folgenden Beispiel:

```
SCREEN 2
.
.
.
'Das Rechteck mißt 141 x 91 Bildpunkte:
GET (10, 10)-(150, 100), Bild
PUT (510, 120), Bild
```

Zum Unterschied von anderen Graphikanweisungen, wie z. B. **LINE** oder **CIRCLE**, schneidet **PUT** außerhalb des Darstellungsfeldes liegende Bilder nicht ab. Stattdessen produziert sie die Fehlermeldung `Unzulässiger Funktionsaufruf.`

Einer der weiteren Vorteile der **PUT**-Anweisung besteht in der Tatsache, daß die Wechselwirkung zwischen dem gespeicherten Abbild und dem bereits existierenden Bildschirminhalt mit dem Argument *Aktionswort* gesteuert werden kann. Das Argument *Aktionswort* kann eine der folgenden Optionen sein: **PSET, PRESET, AND, OR** oder **XOR**.

Wenn der vorhandene Bildschirmhintergrund belanglos ist, verwenden Sie die Option **PSET**, da diese eine exakte Kopie des gespeicherten Abbildes auf den Bildschirm überträgt und alles bereits vorhandene überschreibt.

Tabelle 5.4 zeigt den Einfluß anderer Optionen auf die Art, in der die Anweisung **PUT** die Wechselwirkung zwischen den Bildpunkten eines gespeicherten Bildes und den Bildpunkten auf dem Bildschirm veranlaßt. In dieser Tabelle bedeutet eine 1, daß ein Bildpunkt eingeschaltet, und eine 0, daß ein Bildpunkt ausgeschaltet ist.

Tabelle 5.4 Die Auswirkung unterschiedlicher Funktionsoptionen im Bildschirmmodus 2

Funktionsoption	***Bildpunkt im gespeicherten Bild***	***Bildpunkt auf dem Bildschirm vor der Anweisung PUT***	***Bildpunkt auf dem Bildschirm nach der Anweisung PUT***
PSET	0	0	0
	0	1	0
	1	0	1
	1	1	1
PRESET	0	0	1
	0	1	1
	1	0	0
	1	1	0
AND	0	0	0
	0	1	0
	1	0	0
	1	1	1

Fortsetzung auf der folgenden Seite.

Funktionsoption	*Bildpunkt im gespeicherten Bild*	*Bildpunkt auf dem Bildschirm vor der Anweisung PUT*	*Bildpunkt auf dem Bildschirm nach der Anweisung PUT*
OR	0	0	0
	0	1	1
	1	0	1
	1	1	1
XOR	0	0	0
	0	1	1
	1	0	1
	1	1	0

Wie Sie feststellen können, veranlassen diese Optionen eine **PUT**-Anweisung, Bildpunkte auf dieselbe Art zu behandeln, wie logische Operatoren Zahlen behandeln. **PRESET** gleicht dem logischen Operator **NOT**, da es die Bildpunkte des gespeicherten Bildes umkehrt, ungeachtet dessen, was sich auf dem Bildschirm befindet. Die Optionen **AND**, **OR** und **XOR** sind identisch mit den logischen Operatoren gleichen Namens; zum Beispiel hat die logische Operation

```
1 XOR 1
```

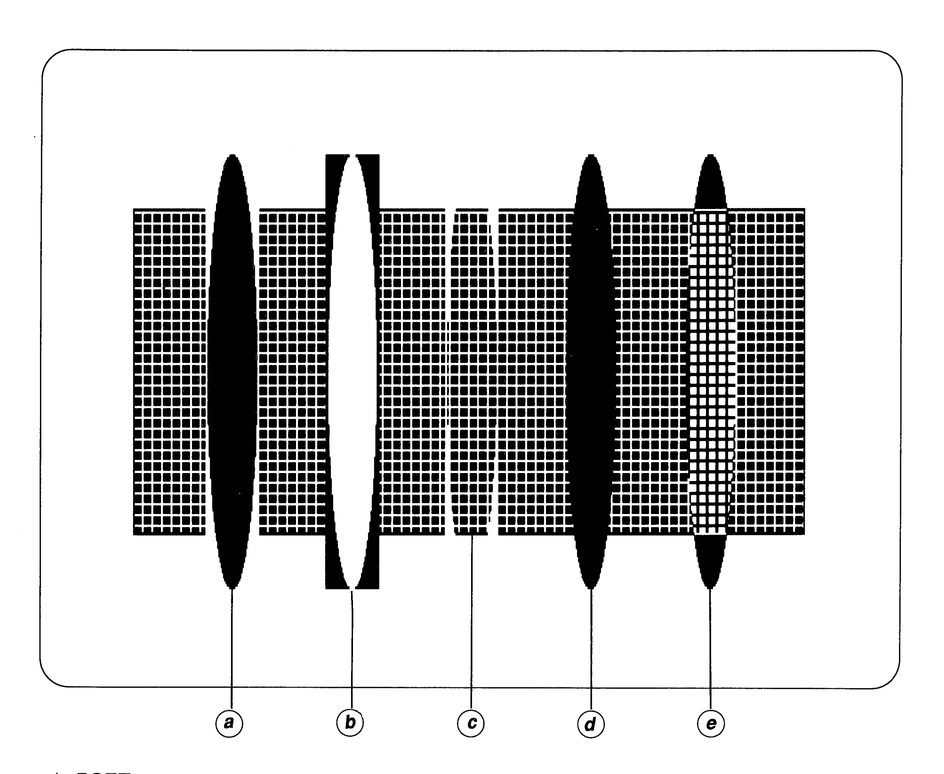
0 als Ergebnis, genauso wie die Verwendung der Option **XOR** einen Bildpunkt ausschaltet, wenn sowohl der Bildpunkt des Abbildes als auch der Bildpunkt des Hintergrundes eingeschaltet sind.

Das folgende Programm überlagert dasselbe Bild fünfmal einem ausgefüllten Rechteck, wobei es jeweils die oben erläuterten Optionen benutzt:

```
SCREEN 2
DIM EllipsBild (1 TO 485) AS INTEGER
' Zeichne und male eine Ellipse aus, speichere dann ihr
' Abbild mit GET:
CIRCLE (22, 100), 80, , , , 4
PAINT (22, 100)
GET (0, 20)-(44, 180), EllipsBild
CLS
' Zeichne ein Rechteck und fülle es mit einem Muster:
LINE (40, 40)-(600, 160), , B
Muster$ = CHR$(126) + CHR$(0) + CHR$(126) + CHR$(126)
PAINT (50, 50), Muster$
```

```
' Lege die Abbilder der Ellipse über das Rechteck unter
' Verwendung unterschiedlicher Funktionsoptionen:
PUT (100, 20), EllipsBild, PSET
PUT (200, 20), EllipsBild, PRESET
PUT (300, 20), EllipsBild, AND
PUT (400, 20), EllipsBild, OR
PUT (500, 20), EllipsBild, XOR
```

Ausgabe



a) PSET
b) PRESET
c) AND
d) OR
e) XOR

In farbunterstützenden Bildschirmmodi bewirken die Optionen **PRESET**, **AND**, **OR** und **XOR** kompliziertere Wechselwirkungen, da die Farbe mehr beinhaltet als einfaches Ein- oder Ausschalten eines Bildpunktes. Die oben aufgestellte Analogie zwischen diesen Optionen und logischen Operatoren gilt jedoch auch in diesen Modi. Wenn zum Beispiel der aktuelle Bildpunkt auf dem Bildschirm die Farbe 1 (kobaltblau in Palette 1) und der Bildpunkt des überlagerten Bildes die Farbe 2 (violett in Palette 1) hat, dann hängt die Farbe des nach einer **PUT**-Anweisung sich ergebenden Bildpunktes von der jeweiligen Option ab, wie in der folgenden Tabelle für nur 2 der 16 unterschiedlichen Kombinationen von Bild- und Hintergrundfarbe gezeigt:

Tabelle 5.5 Die Auswirkungen unterschiedlicher Funktionsoptionen auf die Farbe im Bildschirmmodus 1 (Palette 1)

Funktionsoption	***Bildpunktfarbe im gespeicherten Bild***	***Bildpunktfarbe auf dem Bildschirm vor der PUT-Anweisung***	***Bildschirmfarbe auf dem Bildschirm nach der Anweisung PUT***
PSET	10 (violett)	01 (kobaltblau)	10 (violett)
PRESET	10 (violett)	01 (kobaltblau)	01 (kobaltblau)
AND	10 (violett)	01 (kobaltblau)	00 (schwarz)
OR	10 (violett)	01 (kobaltblau)	11 (weiß)
XOR	10 (violett)	01 (kobaltblau)	11 (weiß)

In Palette 1 ist kobaltblau dem Attribut 1 (01 binär), violett dem Attribut 2 (10 binär) und weiß dem Attribut 3 (11 binär) zugewiesen. Wenn Sie ein EGA besitzen, können Sie den Attributen 1, 2 und 3 unterschiedliche Farben anhand der Anweisung **PALETTE** zuweisen.

5.10.3 Animation mit Hilfe von GET und PUT

Besonders nützlich sind die Anweisungen **GET** und **PUT** für Animation. Die beiden am besten für Animation geeigneten Optionen sind **XOR** und **PSET**. Mit **PSET** ausgeführte Animation ist zwar schneller, löscht aber, wie aus der Ausgabe des letzten Programmes ersichtlich, den Bildschirmhintergrund. **XOR** dagegen ist zwar langsamer, stellt den Bildschirmhintergrund aber nach der Bewegung des Bildes wieder her. Animation mit **XOR** ist wie folgt durchzuführen:

1. Bringen Sie das Objekt mit **XOR** auf den Bildschirm.

2. Berechnen Sie die neue Position des Objektes.
3. Bringen Sie das Objekt ein zweites Mal an der ursprünglichen Stelle auf den Bildschirm und entfernen Sie es anhand von **XOR**.
4. Wiederholen Sie Schritt 1, aber positionieren Sie diesmal das Objekt an den neuen Standort.

Die mit diesen vier Schritten durchgeführte Bewegung läßt den Hintergrund nach Schritt 3 unverändert. Bildflimmern kann verringert werden, wenn die Zeit zwischen den Schritten 4 und 1 minimiert und eine ausreichende Verzögerungszeit zwischen den Schritten 1 und 3 eingebaut wird. Wenn mehr als 1 Objekt animiert werden soll, ist jedes Objekt Schritt für Schritt einzeln zu verarbeiten.

Wenn die Beibehaltung des Hintergrundes unwichtig ist, kann die Option **PSET** für Animation benutzt werden. Der Grundgedanke ist, beim Kopieren des Bildes mit Hilfe der Anweisung **GET**, einen Rahmen um das Bild zu belassen. Wenn dieser Rahmen genauso groß oder größer ist als der maximale Abstand, um den das Objekt sich bewegt, dann entfernt der Rahmen jedesmal, wenn das Bild an eine neue Stelle eingesetzt wird, alle Spuren des Bildes an der alten Position. Diese Methode kann um einiges schneller sein als die oben beschriebene Methode anhand von **XOR**, da für die Bewegung eines Objektes nur eine **PUT**-Anweisung erforderlich ist (obwohl ein größeres Bild bewegt wird).

Beispiel

Das folgende Beispiel zeigt, wie **PUT** mit der Option **PSET** zu verwenden ist, um den Effekt eines vom unteren Rand und den seitlichen Rändern eines Rechtecks zurückprallenden Balles zu erzeugen. Beachten Sie in der folgenden Ausgabe, wie das den Ball enthaltende Rechteck, das in der **GET**-Anweisung angegeben wird, das ausgefüllte Rechteck und die geschriebene Meldung löscht.

Dieses Programm befindet sich in der Datei mit dem Namen *ballpset.bas* auf den QuickBASIC-Originaldisketten.

```
DECLARE FUNCTION FeldGroesse(WLinks, WRechts, WOben, WUnten)

SCREEN 2

' Definiere ein Darstellungsfeld und zeichne hierum
' einen Rahmen:
VIEW (20,10) - (620,190),,1

CONST PI = 3.141592653589#

' Definiere die Koordinaten des Darstellungsfeldes neu
' mit logischen Koordinaten:
WINDOW (-3.15,-.14) - (3.56,1.01)
```

```
' Die Datenfelder im Programm sind nun dynamisch:
' $DYNAMIC

' Berechne die logischen Koordinaten für den oberen
' und unteren Rand eines Rechtecks genügend groß, um das
' Bild aufzunehmen, das mit CIRCLE und PAINT gezeichnet
' wird.

WLinks   = -.21
WRechts  =  .21
WOben    =  .07
WUnten   = -.07

' Rufe die Funktion FeldGroesse auf und übergib ihr
' die logischen Koordinaten des Rechtecks:
FeldGross% = FeldGroesse(WLinks, WRechts, WOben, WUnten)

DIM Feld (1 TO FeldGross%) AS INTEGER

' Zeichne und male den Kreis aus:
CIRCLE (0,0),.18
PAINT (0,0)

' Speichere das Rechteck in Feld:
GET (WLinks,WOben) - (WRechts,WUnten), Feld
CLS

' Zeichne ein Rechteck und fülle es mit einem Muster:
LINE (-3,.8) - (3.4,.2),,B
Muster$ = CHR$(126) + CHR$(0) + CHR$(126) + CHR$(126)
PAINT (0,.5),Muster$

LOCATE 21, 29
PRINT "Beenden mit beliebiger Taste"

' Initialisiere die Schleifenvariablen:
SchrWeit      = .02
StartSchleife = -PI
NimmAb        =  1

DO
   EndSchleife = -StartSchleife
   FOR X = StartSchleife TO EndSchleife STEP SchrWeit

      ' Jedesmal wenn der Ball "aufspringt" (auf dem
      ' unteren Rand des Darstellungsfeldes
      ' auftrifft), wird die Variable NimmAb kleiner,
      ' wodurch der nächste Aufstieg kleiner wird:
      Y = ABS(COS(X)) * NimmAb - .14
      IF Y < -.13 THEN NimmAb = NimmAb * .9
```

```
      ' Beende nach Tastenbetätigung oder wenn NimmAb
      ' kleiner als .01 ist:
      Esc$ = INKEY$
      IF Esc$ <> "" OR NimmAb < .01 THEN EXIT FOR

      ' Bringe das Bild auf den Bildschirm. Der Offset
      ' SchrWeit ist kleiner als der Rahmen um den
      ' Kreis herum. Daher löscht das Bild jedesmal,
      ' wenn es sich bewegt, alle von vorhergehenden
      ' PUT zurückgelassenen Spuren (und löscht ebenso
      ' alles andere auf dem Bildschirm):
      PUT (X,Y), Feld, PSET
   NEXT X

   ' Kehre Richtung um:
   SchrWeit  = -SchrWeit
   StartSchleife = -StartSchleife
LOOP UNTIL Esc$ <> "" OR NimmAb < .01

END

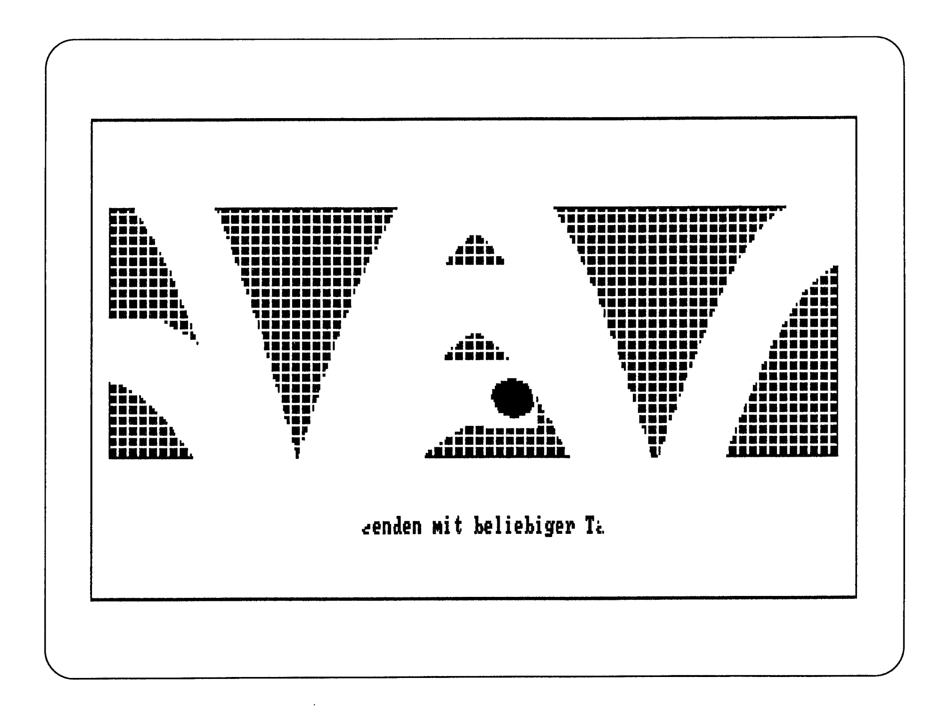

FUNCTION FeldGroesse(WLinks, WRechts, WOben, WUnten) STATIC
   ' Bilde die an diese Funktion übergebenen logischen
   ' Koordinaten auf deren entsprechende physikalische
   ' Koordinaten ab:
   VLinks   = PMAP(WLinks,  0)
   VRechts  = PMAP(WRechts, 0)
   VOben    = PMAP(WOben,   1)
   VUnten   = PMAP(WUnten,  1)

   ' Berechne die Höhe und Breite des umschließenden
   ' Rechtecks in Bildpunkten:
   RechtHoehe = ABS(VUnten - VOben) + 1
   RechtBreit = ABS(VRechts - VLinks) + 1

   ' Berechne Größe des Datenfeldes in Bytes:
   ByteGross = 4 + RechtHoehe * INT((RechtBreit + 7) / 8)

   ' Datenfeld ist Ganzzahl, also dividiere Bytes
   ' durch 2:
   FeldGroesse = ByteGross \ 2 + 1
END FUNCTION
```

Ausgabe



Vergleichen Sie das vorhergehende Programm mit dem nächsten Programm, das **PUT** mit **XOR** zur Beibehaltung des Bildschirmhintergrundes benutzt, wie in den vier oben aufgeführten Schritten beschrieben wurde. Das Rechteck, das den Ball enthält, ist kleiner als im vorhergehenden Programm, da es nicht notwendig ist, einen Rahmen zu lassen. In diesem Fall sind zwei **PUT**-Anweisungen erforderlich, eine, um das Bild sichtbar zu machen, und eine weitere, um es verschwinden zu lassen. Die leere **FOR...NEXT**-Schleife zwischen den **PUT**-Anweisungen verringert das Flimmern, das seine Ursache im zu schnellen Erscheinen und Verschwinden des Bildes hat.

Dieses Programm befindet sich in der Datei mit dem Namen *ballxor.bas* auf den QuickBASIC-Originaldisketten.

```
.
.
.
' Das Rechteck ist kleiner als das im vorhergehenden
' Programm, daher ist auch Feld kleiner:
WLinks   = -.18
WRechts  =  .18
WOben    =  .05
WUnten   = -.05
.
.
.
DO
   EndSchleife = -StartSchleife
   FOR X = StartSchleife TO EndSchleife STEP SchrWeit
      Y = ABS(COS(X)) * NimmAb - .14

      ' Die erste PUT-Anweisung plaziert das Bild auf
      ' dem Bildschirm:
      PUT (X,Y), Feld, XOR

      ' Verwende eine leere FOR...NEXT-Schleife, um
      ' das Programm zu verzögern und das Bildflimmern
      ' zu verringern:
      FOR I = 1 TO 5 : NEXT I

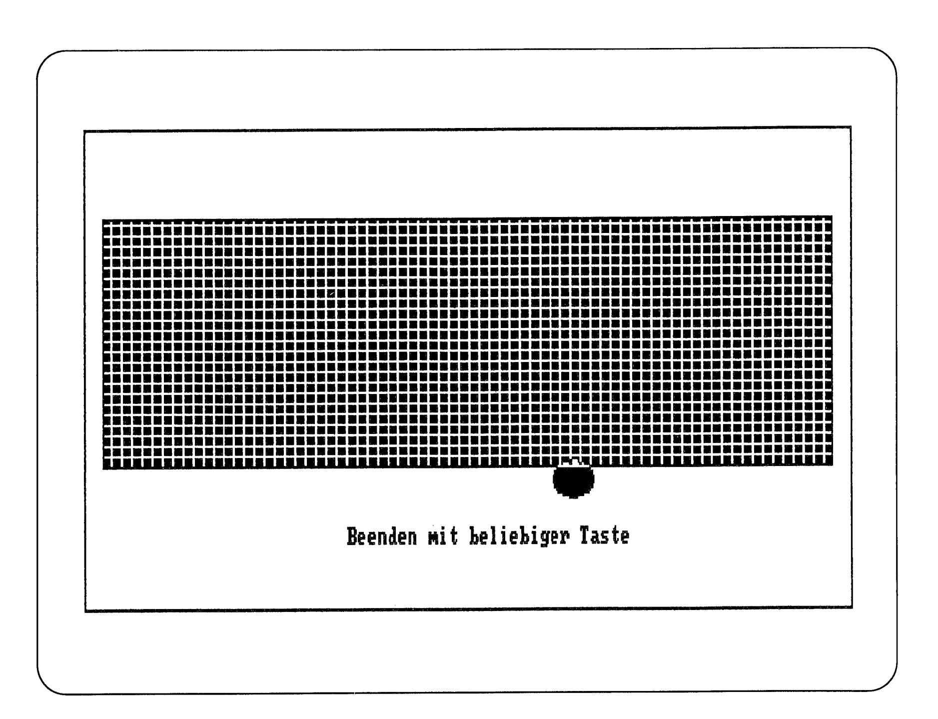
      IF Y < -.13 THEN NimmAb = NimmAb * .9
      Esc$ = INKEY$
      IF Esc$ <> "" OR NimmAb < .01 THEN EXIT FOR

      ' Die zweite PUT-Anweisung löscht das Bild und
      ' stellt den Hintergrund wieder her:
      PUT (X,Y), Feld, XOR
   NEXT X

   SchrWeit  = -SchrWeit
   StartSchleife = -StartSchleife
LOOP UNTIL Esc$ <> "" OR NimmAb < .01

END
.
.
.
```

Ausgabe



5.10.4 Animation mit Bildschirmseiten

Dieser Abschnitt beschreibt eine Technik der Animation, die mehrere Seiten des Videospeichers Ihres Computers ausnutzt.

Seiten im Videospeicher sind mit den Seiten in einem Buch vergleichbar. Je nach den graphischen Fähigkeiten Ihres Computers kann es sein, daß die Bildschirmanzeige nur ein Teil des verfügbaren Videospeichers darstellt - genauso wie die aufgeschlagenen Buchseiten nur ein Teil des Buches bilden. Anders als bei einem Buch können die unsichtbaren Seiten des Videospeichers Ihres Computers jedoch aktiv sein; das heißt, während Sie sich eine Seite auf dem Bildschirm ansehen, kann eine graphische Ausgabe auf den anderen Seiten stattfinden. Das ist so, als wenn der Verfasser an einem Buch weiterschreibt, während Sie es bereits lesen.

Der auf dem Bildschirm sichtbare Bereich des Videospeichers wird als "sichtbare Seite" bezeichnet, während der Bereich des Videospeichers, in den graphische Anweisungen ihre Ausgabe schreiben, "aktive Seite" genannt wird. Die **SCREEN**-Anweisung erlaubt die Auswahl der sichtbaren bzw. aktiven Bildschirmseiten anhand folgender Syntax:

SCREEN [*Modus*],[,[*A-Seite*][, *S-Seite*]]

In dieser Schreibweise ist *A-Seite* die Nummer der aktiven Seite und *S-Seite* die Nummer der sichtbaren Seite. Die aktive und die sichtbare Seite können ein- und dieselbe sein (und sind es standardmäßig, wenn die Argumente *A-Seite* und *S-Seite* nicht mit **SCREEN** verwendet werden; in diesem Fall sind die Werte beider Argumente 0).

Sie können Objekte auf dem Bildschirm bewegen, indem Sie einen Bildschirmmodus mit mehr als einer Seite Videospeicher auswählen, dann die Seiten wechseln und die Ausgabe auf eine oder mehrere aktive Seiten senden, während eine abgeschlossene Ausgabe auf dem Bildschirm angezeigt wird. Diese Technik läßt eine aktive Seite nur sichtbar werden, wenn die Ausgabe auf dieser Seite komplett ist. Da der Betrachter nur ein abgeschlossenes Bild sieht, erscheint die Anzeige augenblicklich.

Beispiel

Das folgende Programm demonstriert die oben erläuterte Technik. Es wählt den aus zwei Seiten bestehenden Bildschirmmodus 7 aus und zeichnet dann mit Hilfe der Anweisung **DRAW** einen Würfel. Dieser Würfel wird dann schrittweise um jeweils 15° gedreht, indem der Wert des **TA**-Makros in der von **DRAW** verwendeten Zeichenkette verändert wird. Durch Hin- und Herschalten der aktiven und sichtbaren Seiten zeigt dieses Programm immer einen vollständigen Würfel, während ein neuer gezeichnet wird.

Dieses Programm befindet sich in der Datei mit dem Namen *kubus.bas* auf den QuickBASIC-Originaldisketten.

```
' Definiere die Makro-Zeichenkette, die zum Zeichnen
' des Würfels sowie Ausmalen seiner Seiten verwendet
' wird:
Plot$ = "BR30 BU25 C1 R54 U45 L54 D45 BE20 P1,1 G20 C2 " +_
  "G20 R54 E20 L54 BD5 P2,2 U5 C4 G20 U45 E20 D45 BL5 P4,4"

AktivSeite% = 1   ' Initialisiere Werte für die
SichtSeite% = 0   ' aktiven und sichtbaren Seiten
Winkel%     = 0   ' sowie für den Rotationswinkel.

DO

   ' Zeichne auf der aktiven Seite, während die
   ' sichtbare Seite gezeigt wird.
   SCREEN 7, , AktivSeite%, SichtSeite%

   CLS 1      ' Lösche die aktive Seite.

   ' Drehe den Würfel "Winkel%" Grad:
   DRAW "TA" + STR$(Winkel%) + Plot$
```

```
    ' Winkel% ist ein Vielfaches von 15 Grad:
    Winkel% = (Winkel% + 15) MOD 360

    ' Das Zeichnen ist abgeschlossen, also lasse den
    ' Würfel in seiner neuen Position sichtbar werden,
    ' durch Vertauschung der aktiven mit der sichtbaren
    ' Seite:
    SWAP AktivSeite%, SichtSeite%
LOOP WHILE INKEY$ = ""        ' Eine Tastenbetätigung
                              ' beendet das Programm.

END
```

5.11 Beispielanwendungen

Die Beispielanwendungen in diesem Kapitel bestehen aus einem Balkendiagramm-Generator, einem Programm, das mit unterschiedlichen Farben Punkte in die Mandelbrot-Menge einzeichnet, und einem Editor für Muster.

5.11.1 Balkendiagramm-Generator (*balken.bas*)

Dieses Programm verwendet zum Zeichnen eines gefüllten Balkendiagramms alle Formen der **LINE**-Anweisung, die in den Abschnitten 5.3.2.1. bis 5.3.2.3 vorgestellt wurden. Jeder Balken wird mit dem in einer **PAINT**-Anweisung festgelegten Muster ausgefüllt. Die Eingabe des Programmes umfaßt Überschriften für das Diagramm, Bezeichnungen für die *x*- und *y*-Achsen sowie einen Satz von höchstens 5 Bezeichnungen (mit den zugeordneten Werten) für die Balken.

Verwendete Anweisungen und Funktionen

Dieses Programm verdeutlicht die Verwendung folgender Graphikanweisungen:

- **LINE**
- **PAINT** (mit einem Muster)
- **SCREEN**

Programm-Listing

Nachstehend wird das Programm für den Balkendiagramm-Generator, *balken.bas*, aufgelistet.

```
' Definiere den Typ für die Titel:
TYPE TitelTyp
   HauptTitel AS STRING * 40
   XTitel AS STRING * 40
   YTitel AS STRING * 18
END TYPE

DECLARE SUB EingabTitel (T AS TitelTyp)
DECLARE FUNCTION Diagramm$ (T AS TitelTyp, Bezeich$(),
Wert!(), N%)
DECLARE FUNCTION EingabDaten% (Bezeich$(), Wert!())

' Variablendeklarationen für Titel und Balkendaten:
DIM Titel AS TitelTyp, Bezeich$(1 TO 5), Wert(1 TO 5)

CONST FALSCH = 0, WAHR = NOT FALSCH

DO
   EingabTitel Titel
   N% = EingabDaten%(Bezeich$(), Wert())
   IF N% <> FALSCH THEN
      NeuDiagr$ = Diagramm$(Titel, Bezeich$(), Wert(), N%)
   END IF
LOOP WHILE NeuDiagr$ = "J"

END
'
' ======================= DIAGRAMM =========================
'   Zeichnet ein Balkendiagramm für die in den
'   Prozeduren EINGABTITEL und EINGABDATEN
'   eingegebenen Daten.
' ==========================================================
'
FUNCTION Diagramm$ (T AS TitelTyp, Bezeich$(), Wert(), N%)_
STATIC

   ' Setze die Größe des Diagramms:
   CONST GRAPHOBEN = 24, GRAPHUNTEN = 171
   CONST GRAPHLINKS = 48, GRAPHRECHTS = 624
   CONST YLANG = GRAPHUNTEN - GRAPHOBEN

   ' Berechne maximalen und minimalen Wert:
   YMax = 0
   YMin = 0
```

```
FOR I% = 1 TO N%
   IF Wert(I%) < YMin THEN YMin = Wert(I%)
   IF Wert(I%) > YMax THEN YMax = Wert(I%)
NEXT I%
' Berechne Breite der Balken und Platz zwischen
' diesen:
BalkBreit = (GRAPHRECHTS - GRAPHLINKS) / N%
BalkZwisch = .2 * BalkBreit
BalkBreit = BalkBreit - BalkZwisch
SCREEN 2
CLS
' Zeichne die y-Achse:
LINE (GRAPHLINKS, GRAPHOBEN)-(GRAPHLINKS, GRAPHUNTEN), 1
' Zeichne Hauptüberschrift des Diagramms:
Start% = 44 - (LEN(RTRIM$(T.HauptTitel)) / 2)
LOCATE 2, Start%
PRINT RTRIM$(T.HauptTitel);
' Bezeichne y-Achse:
Start% = CINT(13 - LEN(RTRIM$(T.YTitel)) / 2)
FOR I% = 1 TO LEN(RTRIM$(T.YTitel))
   LOCATE Start% + I% - 1, 1
   PRINT MID$(T.YTitel, I%, 1);
NEXT I%
' Berechne den Skalierungsfaktor, so daß
' Bezeichnungen nicht größer als 4 Ziffern sind:
IF ABS(YMax) > ABS(YMin) THEN
   Potenz = YMax
ELSE
   Potenz = YMin
END IF
Potenz = CINT(LOG(ABS(Potenz) / 100) / LOG(10))
IF Potenz < 0 THEN Potenz = 0
' Verkleinere minimalen und maximalen Wert:
SkalFaktor = 10 ^ Potenz
YMax = CINT(YMax / SkalFaktor)
YMin = CINT(YMin / SkalFaktor)
' Wenn Potenz nicht Null ist, dann schreibe den
' Skalierungsfaktor in das Diagramm:
IF Potenz <> 0 THEN
   LOCATE 3, 2
   PRINT "x 10^"; LTRIM$(STR$(Potenz))
```

```
END IF
' Setze Markierungsstrich und Zahl für maximalen
' Punkt an die y-Achse:
LINE (GRAPHLINKS - 3, GRAPHOBEN)-STEP(3, 0)
LOCATE 4, 2
PRINT USING "####"; YMax
' Setze Markierungsstrich und Zahl für minimalen
' Punkt an die y-Achse:
LINE (GRAPHLINKS - 3, GRAPHUNTEN) - STEP(3, 0)
LOCATE 22, 2
PRINT USING "####"; YMin
YMax = YMax * SkalFaktor ' Vergrößere Minimum und
                         ' Maximum wieder
YMin = YMin * SkalFaktor ' zur grafischen
                         ' Darstellung der
                         ' Berechnungen.
' Bezeichne x-Achse:
Start% = 44 - (LEN(RTRIM$(T.XTitel)) / 2)
LOCATE 25, Start%
PRINT RTRIM$(T.XTitel);
' Berechne den Bildpunktebereich für die y-Achse:
YBereich = YMax - YMin
' Definiere ein diagonal gestreiftes Muster:
Kachel$ = CHR$(1) + CHR$(2) + CHR$(4) + CHR$(8) + _
          CHR$(16) + CHR$(32) + CHR$(64) + CHR$(128)
' Zeichne eine Null-Linie, falls passend:
IF YMin < 0 THEN
   Unten = GRAPHUNTEN - ((-YMin) / YBereich * YLANG)
   LOCATE INT((Unten - 1) / 8) + 1, 5
   PRINT "0";
ELSE
   Unten = GRAPHUNTEN
END IF
' Zeichne die x-Achse:
LINE (GRAPHLINKS - 3, Unten)-(GRAPHRECHTS, Unten)
' Zeichne die Balken und Bezeichnungen:
Start% = GRAPHLINKS + (BalkZwisch / 2)
FOR I% = 1 TO N%
```

```
      ' Zeichne eine Balkenbezeichnung:
      BalkMitt = Start% + (BalkBreit / 2)
      DiagrMitt = INT((BalkMitt - 1) / 8) + 1
      LOCATE 23,DiagrMitt - INT(LEN(RTRIM$(Bezeich$(I%)))/2)
      PRINT Bezeich$(I%);

      ' Zeichne den Balken und fülle ihn mit dem
      ' gestreiften Muster aus:
      BalkHoehe = (Wert(I%) / YBereich) * YLANG
      LINE (Start%, Unten)-STEP(BalkBreit, - BalkHoehe),,B
      PAINT (BalkMitt, Unten - (BalkHoehe / 2)),Kachel$,1
      Start% = Start% + BalkBreit + BalkZwisch
   NEXT I%
   LOCATE 1, 1
   PRINT "Neues Diagramm? ";
   Diagramm$ = UCASE$(INPUT$(1))
END FUNCTION
'
' ===================== EINGABDATEN ========================
'       Liest Eingabe für die Balkenbezeichnungen
'       und deren Werte
' ==========================================================
'
FUNCTION EingabDaten% (Bezeich$(), Wert()) STATIC

   ' Initialisiere die Anzahl der Datenwerte:
   AnzDaten% = 0

   ' Gib Anweisungen für Dateneingabe aus:
   CLS
   PRINT "Geben Sie Daten für bis zu 5 Balken ein:"
   PRINT "   * Geben Sie die Bezeichnung und den Wert";
   PRINT " für jeden Balken ein."
   PRINT "   * Werte können negativ sein."
   PRINT "   * Geben Sie eine leere Bezeichnung zum";
   PRINT " Beenden ein."
   PRINT
   PRINT "Nachdem Sie das Diagramm gesehen haben, ";
   PRINT "betätigen Sie eine Taste"
   PRINT "zum Beenden des Programmes.";

   ' Nimm Daten bis zu einer leeren Bezeichnung
   ' oder 5 Einträgen an:
   Fertig% = FALSCH
   DO
```

```
      AnzDaten% = AnzDaten% + 1
      PRINT
      PRINT "Balken("; LTRIM$(STR$(AnzDaten%)); "):"
      INPUT ; "Bezeichnung? ", Bezeich$(AnzDaten%)

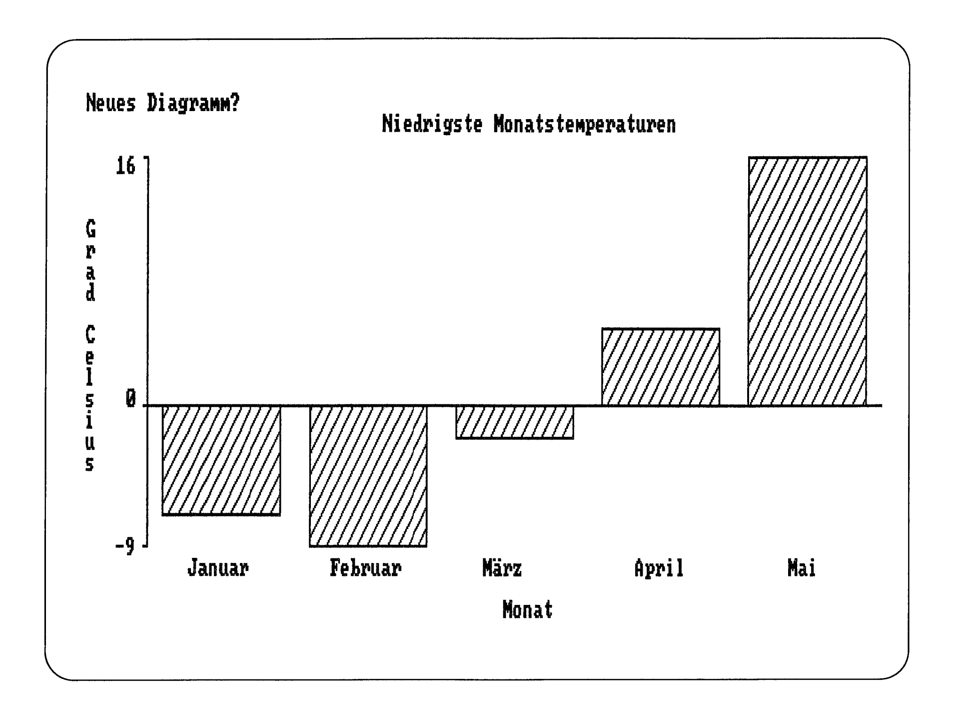
      ' Eingabe eines Wertes nur, wenn Bezeichnung
      ' nicht leer ist:
      IF Bezeich$(AnzDaten%) <> "" THEN
         entf = LEN(Bezeich$(AnzDaten%)) + 30
         LOCATE , entf
         INPUT "Wert? ", Wert(AnzDaten%)

      ' Wenn Bezeichnung leer ist, vermindere den
      ' Datenzähler und setze Flagge für Fertig auf
      ' WAHR:
      ELSE
         AnzDaten% = AnzDaten% - 1
         Fertig% = WAHR
      END IF
   LOOP UNTIL (AnzDaten% = 5) OR Fertig%

   ' Gib die Anzahl der eingegebenen Datenwerte
   ' zurück:
   EingabDaten% = AnzDaten%
END FUNCTION
'
' ==================== EINGABTITEL ========================
'       Akzeptiert Eingabe für die drei
'       unterschiedlichen Diagrammtitel
' =========================================================
'
SUB EingabTitel (T AS TitelTyp) STATIC
   SCREEN 0, 0    ' Setze Textbildschirm.
   DO             ' Eingabe der Titel.
      CLS
      PRINT "Geben Sie die Überschrift des Diagramms ein :";
      INPUT "", T.HauptTitel
      INPUT "Geben Sie die Bezeichnung der x-Achse ein   :";
      INPUT "", T.XTitel
      INPUT "Geben Sie die Bezeichnung der y-Achse ein   :";
      INPUT "", T.YTitel

      ' Überprüfe, ob Titel in Ordnung sind:
      LOCATE 7, 1
      PRINT "Eingaben korrekt? J/N ";
      LOCATE , , 1
      OK$ = UCASE$(INPUT$(1))
   LOOP UNTIL OK$ = "J"
END SUB
```

Ausgabe



5.11.2 Mathematisch erzeugte Farbe in einer Figur (*mandel.bas*)

Dieses Programm verwendet BASIC-Graphikanweisungen, um eine als "Fractal" bezeichnete Figur zu erzeugen. Ein Fractal ist eine graphische Darstellung der Veränderung von Zahlen, die einer wiederholten Folge von mathematischen Operationen unterworfen sind. Der von diesem Programm erzeugte Fractal zeigt eine Teilmenge aus der Menge der sogenannten komplexen Zahlen; diese Teilmenge wurde "Mandelbrot-Menge" nach Benoit B. Mandelbrot vom IBM Thomas J. Watson Research Center benannt.

Im wesentlichen bestehen komplexe Zahlen aus zwei Teilen, einem reellen und einem sogenannten imaginären Teil, der ein Vielfaches von $\sqrt{-1}$ ist. Quadrieren einer komplexen Zahl, anschließendes Zurückgeben der imaginären und reellen Teile in eine zweite komplexe Zahl und Wiederholung dieses Vorganges veranlassen einige komplexe Zahlen, sich sehr schnell zu vergrößern. Andere Zahlen jedoch läßt dieses Verfahren um einen stabilen Wert schwanken. Die stabilen Werte werden in der Mandelbrot-Menge eingegliedert und in diesem Programm durch die Farbe schwarz dargestellt. Die instabilen Werte, also diejenigen, die sich von der Mandelbrot-Menge entfernen, werden durch die anderen Farben der Palette dargestellt. Je kleiner das Farbattribut, desto instabiler der Punkt.

Weitere Hintergrundinformation zu der Mandelbrot-Menge finden Sie im A.K. Dewdneys Aufsatz "Computer Recreations" im *Scientific American*, Ausgabe August 1985.

Dieses Programm überprüft ebenfalls das Vorhandensein einer EGA-Karte und zeichnet in diesem Fall die Mandelbrot-Menge im Bildschirmmodus 8. Nach dem Zeichnen jeder Zeile wechselt das Programm die 16 Farben der Palette mit Hilfe einer **PALETTE USING**-Anweisung. Wenn keine EGA-Karte vorhanden ist, zeichnet das Programm eine 4-farbige (weiß, violett, kobaltblau und schwarz) Mandelbrot-Menge im Bildschirmmodus 1.

Verwendete Anweisungen und Funktionen

Dieses Programm veranschaulicht die Verwendung folgender Graphikanweisungen:

- **LINE**
- **PALETTE USING**
- **PMAP**
- **PSET**
- **SCREEN**
- **VIEW**
- **WINDOW**

Programm-Listing

```
DEFINT A-Z        ' Standardvariablentyp ist Ganzzahl.

DECLARE SUB AenderPalette ()
DECLARE SUB FenstWerte (FL%, FR%, FO%, FU%)
DECLARE SUB MonitorTest (EM%, FB%, DL%, DR%, DOB%, DU%)

CONST FALSCH = 0, WAHR = NOT FALSCH  ' Boolesche
                                     ' Konstanten

' Lege maximale Anzahl an Iterationen pro Punkt fest:
CONST MAXSCHLEIFE = 30, MAXGROSS = 1000000

DIM PaletteFeld(15)
FOR I = 0 TO 15 : PaletteFeld(I) = I : NEXT I

' Rufe FenstWerte auf, um die Koordinaten der Ecken des
' Fensters zu bekommen:
FenstWerte WLinks, WRechts, WOben, WUnten
```

```
' Rufe MonitorTest auf um herauszufinden, ob es sich
' um einen Rechner mit EGA handelt, und lies
' Koordinaten der Ecken des Darstellungsfeldes:
MonitorTest EgaModus, FarbeBereich, VLinks, VRechts, _
            VOben, VUnten

' Definiere Darstellungsfeld und entsprechendes
' Fenster:
VIEW (VLinks, VOben)-(VRechts, VUnten), 0, FarbeBereich
WINDOW (WLinks, WOben)-(WRechts, WUnten)

LOCATE 24, 10 : PRINT "Beliebige Taste zum Beenden.";

XLaenge = VRechts - VLinks
YLaenge = VUnten - VOben
FarbeBreit = MAXSCHLEIFE \ FarbeBereich

' Bearbeite jeden Bildpunkt im Darstellungsfeld und
' berechne, ob er sich in der Mandelbrot-Menge
' befindet oder nicht:
FOR Y = 0 TO YLaenge   ' Durchlaufe jede Zeile des
                       ' Darstellungsfeldes.
   LogikY = PMAP(Y, 3) ' Lies die logische
                       ' y-Koordinate des Bildpunktes.
   PSET (WLinks, LogikY)  ' Zeichne den äußerst linken
                          ' Bildpunkt einer Zeile.
   AltFarbe = 0    ' Beginne mit der Hintergrundfarbe.

   FOR X = 0 TO XLaenge   ' Durchlaufe jeden Bildpunkt
                          ' einer Zeile.
      LogikX = PMAP(X, 2) ' Lies die logische
                          ' x-Koordinate des
                          ' Bildpunktes.
      MandelX& = LogikX
      MandelY& = LogikY

      ' Führe die Berechnungen aus, um festzustellen,
      ' ob dieser Punkt sich in der Mandelbrot-Menge
      ' befindet.

      FOR I = 1 TO MAXSCHLEIFE
         RealZahl& = MandelX& * MandelX&
         ImagZahl& = MandelY& * MandelY&
         IF (RealZahl& +ImagZahl&) >= MAXGROSS THEN EXIT FOR
         MandelY& = (MandelX& * MandelY&) \ 250 + LogikY
         MandelX& = (RealZahl& - ImagZahl&) \ 500 + LogikX
      NEXT I
```

```
        ' Weise dem Punkt eine Farbe zu:
        PFarbe = I \ FarbeBreit

        ' Wenn die Farbe gewechselt hat, zeichne eine
        ' Zeile von dem letzten Punkt, auf den Bezug
        ' genommen wurde, zu dem neuen Punkt und
        ' verwende die alte Farbe.
        IF PFarbe  <> AltFarbe THEN
          LINE -(LogikX, LogikY), (FarbeBereich - AltFarbe)
          AltFarbe = PFarbe
        END IF

        IF INKEY$ <> "" THEN END
   NEXT X

   ' Zeichne den letzten Zeilenabschnitt zur rechten
   ' Seite des Darstellungsfeldes:
   LINE -(LogikX, LogikY), (FarbeBereich - AltFarbe)

   ' Wenn es sich um eine EGA-Maschine handelt, ändere
   ' die Palette nach dem Zeichnen jeder Zeile:
   IF EgaModus THEN AenderPalette
NEXT Y

DO
   ' Verändere die Palette solange, bis der Benutzer
   ' eine Taste betätigt:
   IF EgaModus THEN AenderPalette
LOOP WHILE INKEY$ = ""

SCREEN 0, 0        ' Schalte den Bildschirm wieder auf
WIDTH 80           ' den Textmodus, 80 Spalten.
END

FalschMonit:           ' Fehlerbehandlung, die aufgeru-
   EgaModus = FALSCH   ' fen wird, wenn keine EGA-
   RESUME NEXT         ' Graphikkarte vorhanden ist.
'
' ==================== AenderPalette =======================
'      Verschiebt die Palette bei jedem Aufruf um eins
' ==========================================================
'
SUB AenderPalette STATIC
   SHARED PaletteFeld(), FarbeBereich

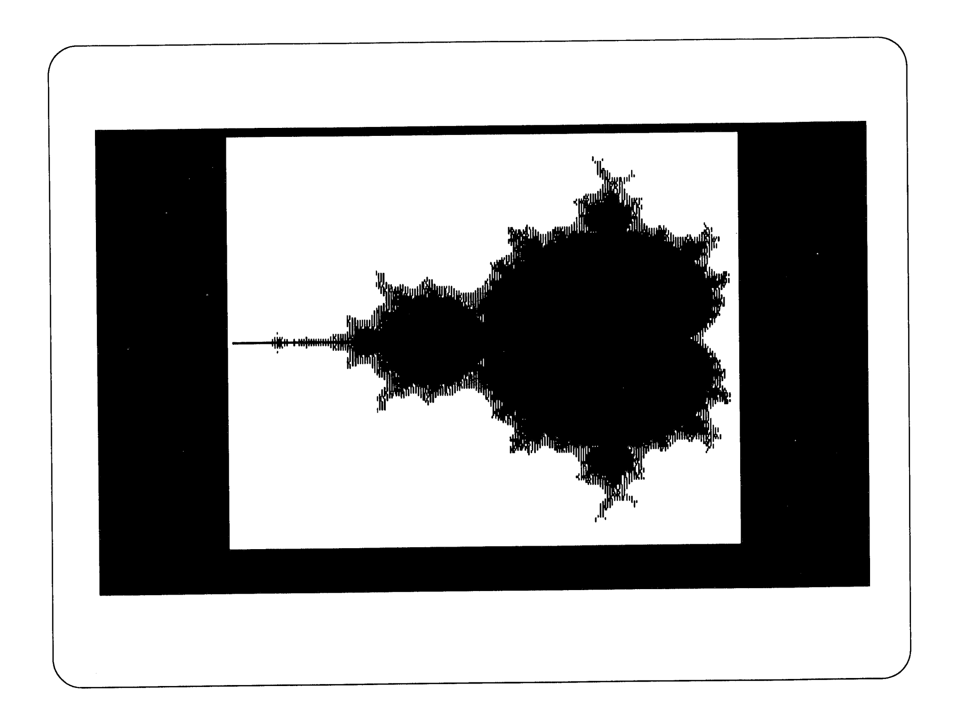
   FOR I = 1 TO FarbeBereich
      PaletteFeld(I) = (PaletteFeld(I) MOD FarbeBereich) + 1
   NEXT I
   PALETTE USING PaletteFeld(0)
```

```
END SUB
'
' =================== MonitorTest ====================
' Verwendet eine Anweisung SCREEN 8 als Prüfung, ob
' der Benutzer EGA-Hardware einsetzt. Wenn sich dabei
' ein Fehler ergibt, wird die EM-Flagge auf FALSCH
' gesetzt, und der Bildschirm wird mit SCREEN 1
' gesetzt.
'
' Setzt ebenso Werte für die Ecken des
' Darstellungsfeldes (DL = Links, DR = Rechts,
' DOB = Oben, DU = Unten), skaliert mit dem
' richtigen Bildverhältnis, so daß das
' Darstellungsfeld wirklich ein Quadrat ist.
' ====================================================
'
SUB MonitorTest (EM, FB, DL, DR, DOB, DU) STATIC
   EM = WAHR
   ON ERROR GOTO FalschMonit
   SCREEN 8, 1
   ON ERROR GOTO 0
   IF EM THEN      ' Kein Fehler, also ist SCREEN 8 OK.
      DL = 110 : DR = 529
      DOB = 5 : DU = 179
      FB = 15      ' 16 Farben (0 - 15)
   ELSE            ' Fehler, also verwende SCREEN 1.
      SCREEN 1, 1
      DL = 55 : DR = 264
      DOB = 5 : DU = 179
      FB = 3       ' 4 Farben (0 - 3)
   END IF
END SUB
'
' ================== FenstWerte =====================
' Liest Fensterecken als Benutzereingabe oder setzt
' die Werte der Ecken, wenn keine Eingabe vorhanden
' ist.
' ====================================================
'
SUB FenstWerte (FL, FR, FO, FU) STATIC
   CLS
```

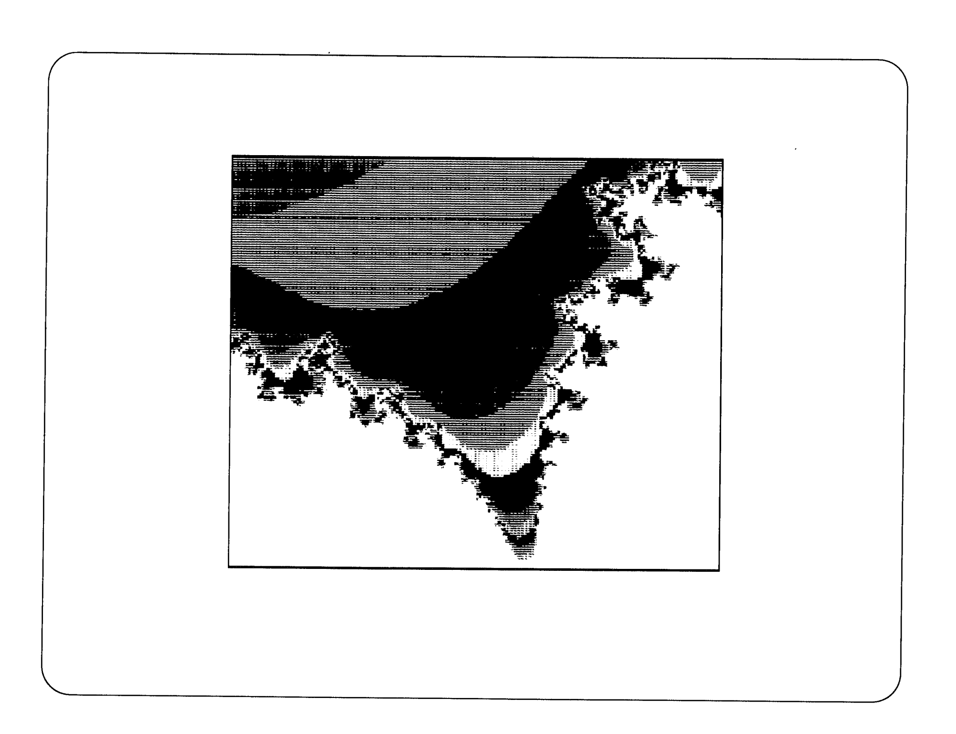
```
    PRINT "Dieses Programm druckt die grafische ";
    PRINT "Darstellung der vollständigen Mandelbrot-Menge."
    PRINT "Das Standardfenster reicht von (-1000,625) bis";
    PRINT " (250,-625)."
    PRINT " Zum Vergrößern eines Teils der Figur geben Sie";
    PRINT " Koordinaten innerhalb dieses Fensters ein."
    PRINT
    PRINT "Nach Betätigen von <EINGABETASTE> sehen Sie ";
    PRINT " das Standardfenster."
    PRINT "Betätigen Sie eine andere Taste, um eigene ";
    PRINT "Fensterkoordinaten einzugeben:";
    LOCATE , , 1
    Antw$ = INPUT$(1)
    ' Benutzer hat EINGABETASTE nicht betätigt, also
    ' lies Fensterecken ein:
    IF Antw$ <> CHR$(13) THEN
       PRINT
       INPUT "X-Koordinate der oberen linken Ecke: ", FL
       DO
          INPUT "X-Koordinate der unteren rechten Ecke: ", FR
          IF FR <= FL THEN
             PRINT "Rechte Ecke muß größer als linke Ecke_
                    sein."
          END IF
       LOOP WHILE FR <= FL
       INPUT "Y-Koordinate der oberen linken Ecke: ", FO
       DO
          INPUT "Y-Koordinate der unteren rechten Ecke: ", FU
          IF FU >= FO THEN
             PRINT "Untere Ecke muß kleiner als obere Ecke_
                    sein."
          END IF
       LOOP WHILE FU >= FO
    ' Benutzer betätigte EINGABETASTE, also setze
    ' Standardwerte:
    ELSE
       FL = -1000
       FR =   250
       FO =   625
       FU =  -625
    END IF
END SUB
```

Ausgabe

Die folgende Abbildung zeigt die Mandelbrot-Figur im Bildschirmmodus 1. Diese Ausgabe ist beim Vorhandensein einer CGA zu sehen, wenn die Koordinaten des Standardfensters gewählt werden.



Die nächste Abbildung zeigt die Mandelbrot-Figur mit (*x*,*y*)-Koordinaten von `(-500, 250)` für die obere linke Ecke und `(-300, 50)` für die untere rechte Ecke. Diese Abbildung ist im Bildschirmmodus 8 gezeichnet, den das Programm wählt, wenn Sie über einen EGA oder VGA verfügen.



5.11.3 Editor für Muster (*edmust.bas*)

Dieses Programm ermöglicht die Editierung einer Musterkachel für die Verwendung mit **PAINT**. Während Sie die Kachel auf der linken Seite des Bildschirms editieren, ist das Erscheinungsbild des fertigen Musters auf der rechten Seite des Bildschirms sichtbar. Ist das Editieren der Musterkachel beendet, gibt das Programm die ganzzahligen Argumente aus, die die Anweisung **CHR$** zum Zeichnen jeder Zeile der Kachel verwendet.

Verwendete Anweisungen und Funktionen

Dieses Programm veranschaulicht die Verwendung folgender Graphikanweisungen:

- **LINE**
- **PAINT** (mit Muster)
- **VIEW**

Programm-Listing

```
DECLARE SUB ZeichneMuster ()
DECLARE SUB EditMuster    ()
DECLARE SUB Initialisiere ()
DECLARE SUB ZeigMuster    (OK$)

DIM Bit%(0 TO 7), Muster$, Esc$, MusterGross%

DO
   Initialisiere
   EditMuster
   ZeigMuster OK$
LOOP WHILE OK$ = "J"

END
'
' ===================== ZEICHNEMUSTER ======================
'        Zeichnet ein gemustertes Rechteck auf der
'        rechten Seite des Bildschirms.
' ==========================================================
'
SUB ZeichneMuster STATIC
SHARED Muster$

   VIEW (320, 24)-(622, 160), 0, 1 ' Setze Feld auf
                                   ' Rechteck.

   PAINT (1, 1), Muster$     ' Verwende PAINT, um es
                             ' auszufüllen.

   VIEW                      ' Setze Feld auf gesamten
                             ' Bildschirm.

END SUB
'
' ======================= EDITMUSTER =======================
'              Editiert ein Kachel-Bytemuster
' ==========================================================
'
SUB EditMuster STATIC
SHARED Muster$, Esc$, Bit%(), MusterGross%

   ByteZahl% = 1    ' Anfangsposition.
   BitZahl% = 7
   Null$ = CHR$(0)  ' CHR$(0) ist das erste zurückgegebene
                    ' Byte der 2-Byte-Zeichenkette, wenn
                    ' eine Richtungstaste wie OBEN oder
                    ' UNTEN betätigt wurde.
```

```
DO
   ' Berechne Anfangsposition dieses Bits auf dem
   ' Bildschirm:
   X% = ((7 - BitZahl%) * 16) + 80
   Y% = (ByteZahl% + 2) * 8

   ' Warte auf Tastenbetätigung (und lasse Cursor
   ' alle 3/10 Sekunden blinken):
   Zustand% = 0
   AufZeit = 0
   DO

   ' Prüfe Zeitgeber und wechsle Cursor-Zustand,
   ' wenn 3/10 Sekunden vergangen sind:
   IF ABS(TIMER - AufZeit) > .3 THEN
      AufZeit = TIMER
      Zustand% = 1 - Zustand%

      ' Schalte Rahmen des Bits an und aus:
      LINE (X%-1, Y%-1)-STEP(15, 8), Zustand%, B
   END IF

   Pruef$ = INKEY$    ' Prüfe auf Tastenbetätigung.

   LOOP WHILE Pruef$ = "" ' Durchlaufe Schleife,
                          ' bis eine Taste betätigt
                          ' ist.

   ' Lösche Cursor:
   LINE (X%-1, Y%-1)-STEP(15, 8), 0, B

   SELECT CASE Pruef$   ' Reagiere auf
                        ' Tastenbetätigung.

      CASE CHR$(27)     ' ESC-Taste betätigt:
         EXIT SUB       ' verlasse dieses
                        ' Unterprogramm.

      CASE CHR$(32)     ' LEERTASTE betätigt:
                        ' setze Zustand des Bits
                        ' neu.

         ' Invertiere Bit in der
         ' Musterzeichenkette:
         AktuellByte% = ASC(MID$(Muster$, ByteZahl%, 1))
         AktuellByte% = AktuellByte% XOR Bit%(BitZahl%)
         MID$ (Muster$, ByteZahl%) = CHR$(AktuellByte%)
```

```
            ' Zeichne Bit neu auf den Bildschirm:
            IF (AktuellByte% AND Bit%(BitZahl%)) <> 0 THEN
               AktuellFarbe% = 1
            ELSE
               AktuellFarbe% = 0
            END IF
            LINE (X%+1, Y%+1)-STEP(11, 4), AktuellFarbe%, BF
         CASE CHR$(13)       ' EINGABETASTE betätigt:
            ZeichneMuster    ' zeichne Muster im rechten
                             ' Rechteck.
         CASE Null$ + CHR$(75)  ' LINKS-Taste: bewege
                                ' Cursor nach links.

            BitZahl% = BitZahl% + 1
            IF BitZahl% > 7 THEN BitZahl% = 0
         CASE Null$ + CHR$(77)  ' RECHTS-Taste: bewege
                                ' Cursor nach rechts.

            BitZahl% = BitZahl% - 1
            IF BitZahl% < 0 THEN BitZahl% = 7
         CASE Null$ + CHR$(72)  ' OBEN-Taste: bewege
                                ' Cursor nach oben.

            ByteZahl% = ByteZahl% - 1
            IF ByteZahl% < 1 THEN
               ByteZahl% = MusterGross%
            END IF
         CASE Null$ + CHR$(80)  ' UNTEN-Taste: bewege
                                ' Cursor nach unten.

            ByteZahl% = ByteZahl% + 1
            IF ByteZahl% > MusterGross% THEN ByteZahl% = 1
         CASE ELSE
            ' Benutzer hat eine andere Taste als ESC,
            ' LEERTASTE, EINGABETASTE, OBEN, UNTEN,
            ' LINKS oder RECHTS betätigt, also tue
            ' nichts.
      END SELECT
   LOOP
END SUB
```

```
'
' ===================== INITIALISIERE ======================
'        Initialisiert Anfangsmuster und Bildschirm.
' ==========================================================
'
SUB Initialisiere STATIC
SHARED Muster$, Esc$, Bit%(), MusterGross%
   Esc$ = CHR$(27)    ' ESC-Zeichen ist ASCII 27.

   ' Initialisiere ein Datenfeld, das Bits in den
   ' Positionen 0 bis 7 enthält:
   FOR I% = 0 TO 7
      Bit%(I%) = 2 ^ I%
   NEXT I%

   CLS

   ' Eingabe der Größe des Musters (in Anzahl an
   ' Bytes):
   LOCATE 5, 5
   PRINT "Geben Sie die Größe des Musters ein ";
   PRINT "(1-16 Zeilen):";
   DO
      LOCATE 5, 58
      PRINT "    ";
      LOCATE 5, 58
      INPUT "", MusterGross%
   LOOP WHILE MusterGross% < 1 OR MusterGross% > 16

   ' Setze Anfangsmuster auf alle Bits gesetzt:
   Muster$ = STRING$(MusterGross%, 255)

   SCREEN 2    ' 640 x 200 monochromer Grafikmodus

   ' Zeichne Trennungslinien:
   LINE (0, 10)-(635, 10),1
   LINE (300, 0)-(300, 199)
   LINE (302, 0)-(302, 199)

   ' Schreibe Überschriften:
   LOCATE 1, 13: PRINT "Musterbytes"
   LOCATE 1, 53: PRINT "Musteransicht"

   ' Zeichne Editierbildschirm für das Muster:
   FOR I% = 1 TO MusterGross%

      ' Gib links von jeder Zeile Bezeichnung aus:
      LOCATE I% + 3, 8
      PRINT USING "##:"; I%
```

```
      ' Zeichne "Bit"-Rechtecke:
      X% = 80
      Y% = (I% + 2) * 8
      FOR J% = 1 TO 8
        LINE (X%, Y%)-STEP(13, 6), 1, BF
        X% = X% + 16
      NEXT J%
   NEXT I%

   ZeichneMuster   ' Zeichne "Musteransicht"-Rechteck.

   LOCATE 21, 1
   PRINT "RICHTUNGSTASTEN........Bewegen Cursor"
   PRINT "LEERTASTE............Verändert Punkt"
   PRINT "EINGABETASTE.....Anzeigen des Musters"
   PRINT "ESC..........................Beendet";
END SUB
'
' ====================== ZEIGMUSTER =======================
'        Gibt die von PAINT zum Erstellen des Musters
'        verwendeten CHR$-Werte
' ==========================================================
'
SUB ZeigMuster (OK$) STATIC
SHARED Muster$, MusterGross%
   ' Schalte Bildschirm zurück in 80-Spalten Textmodus:
   SCREEN 0, 0
   WIDTH 80

   PRINT "Die folgenden Zeichen erstellen Ihr Muster:"
   PRINT

   ' Gib den Wert für jedes Musterbyte aus:
   FOR I% = 1 TO MusterGross%
      MusterByte% = ASC(MID$(Muster$, I%, 1))
      PRINT "CHR$("; LTRIM$(STR$(MusterByte%)); ")"
   NEXT I%

   PRINT
   LOCATE ,,1
   PRINT "Neues Muster? ";
   OK$ = UCASE$(INPUT$(1))
END SUB
```

Ausgabe

Das ist ein von einem Editor für Muster erzeugtes Musterbeispiel.

Musterbytes

Musteransicht

1:
2:
3:
4:
5:
6:
7:
8:
9:
10:

RICHTUNGSTASTEN........Bewegen Cursor
LEERTASTE.............Verändert Punkt
EINGABETASTE.....Anzeigen des Musters
ESC.........................Beendet

6 Fehler- und Ereignisverfolgung

Dieses Kapitel zeigt Ihnen, wie Fehler und Ereignisse verfolgt werden, die während der Programmausführung auftreten. Mit Hilfe der Fehlerverfolgung können Sie das Programm vor Benutzerfehlern schützen, wie zum Beispiel die Eingabe einer Zeichenkette wenn ein numerischer Wert erwartet wird, oder der Versuch, eine Datei in einem nicht existierenden Verzeichnis zu öffnen. Die Ereignisverfolgung versetzt Ihr Programm in die Lage, Laufzeitereignisse zu entdecken und zu verarbeiten, wie es bei Tastenbetätigungen oder Dateneingang in einem **COM**-Anschluß der Fall ist.

In diesem Kapitel lernen Sie, wie folgende Aufgaben durchgeführt werden:

- Aktivieren der Fehler- oder Ereignisverfolgung.
- Schreiben einer Routine zur Verarbeitung der verfolgten Fehler oder Ereignisse.
- Zurückgeben der Kontrolle von einer Fehlerbehandlungsroutine oder einer Ereignisverfolgungsroutine.
- Schreiben eines Programmes, das jede Tastenbetätigung bzw. Kombination von Tastenbetätigungen verfolgt.
- Verfolgen von Fehlern oder Ereignissen innerhalb von **SUB**- oder **FUNCTION**-Prozeduren.
- Verfolgen von Fehlern und Ereignissen in Programmen, die aus mehr als einem Modul bestehen.

6.1 Fehlerverfolgung

Die Fehlerverfolgung ermöglicht dem Programm Laufzeitfehler abzufangen, bevor diese den Programmablauf beenden. Ohne Fehlerverfolgung veranlassen Fehler während der Programmausführung (wie zum Beispiel der Versuch eine nicht existierende Datendatei zu öffnen) BASIC, die entsprechende Fehlermeldung auszugeben und das Programm zu beenden. Wenn der Benutzer eine selbständige Version Ihres Programmes verwendet, wird er das Programm neu starten müssen, wobei alle vor Auftreten des Fehlers eingegebenen Daten oder durchgeführten Berechnungen verloren gehen.

Ist die Fehlerverfolgung aktiv, veranlaßt der entdeckte Fehler das Programm zu einer Verzweigung in eine "Fehlerbehandlungsroutine", die den Fehler korrigiert. Wenn Sie in der Lage sind, die möglicherweise beim Benutzen Ihres Programmes auftretenden Fehler abzufangen, bietet Ihnen die Fehlerverfolgung die Möglichkeit, diese Fehler "benutzerfreundlich" zu behandeln.

Abschnitte 6.1.1 und 6.1.2 erläutern, wie die Fehlerverfolgung aktiviert, wie eine Routine zur Behandlung von verfolgten Fehlern geschrieben und wie die Kontrolle von der Fehlerbehandlungsroutine zurückgegeben wird, nachdem diese den Fehler bearbeitet hat.

6.1.1 Aktivierung der Fehlerverfolgung

Die Fehlerverfolgung wird mit der Anweisung **ON ERROR GOTO** *Zeile* aktiviert, wobei *Zeile* eine Zeilennummer oder Zeilenmarke ist, die die erste Zeile einer Fehlerbehandlungsroutine kennzeichnet. Sobald BASIC eine Anweisung **ON ERROR GOTO** *Zeile* vorgefunden hat, verursacht jeder Laufzeitfehler innerhalb des Moduls, der diese Anweisung enthält, eine Verzweigung in die angegebene *Zeile*. (Wenn die Nummer oder Marke nicht innerhalb des Moduls vorhanden ist, erscheint eine Laufzeitfehlermeldung mit dem Text `Nicht definierte Marke`.)

Eine Anweisung **ON ERROR GOTO** muß vor dem Einsatz der Fehlerverfolgung ausgeführt werden. Demnach muß die Anweisung **ON ERROR GOTO** so plaziert werden, daß sie die Programmausführung erreicht, bevor sie benötigt wird. In der Regel ist deshalb diese Anweisung eine der ersten ausführbaren Anweisungen im Hauptmodul oder in einer Prozedur.

Sie können eine Anweisung **ON ERROR GOTO 0** nicht zur Verzweigung in eine Fehlerbehandlung verwenden, da diese eine besondere Bedeutung für die Fehlerverfolgung hat. Die Anweisung **ON ERROR GOTO 0** wirkt sich, je nach der Stelle an der sie erscheint, auf zwei Arten aus. Wenn sie außerhalb einer Fehlerbehandlungsroutine erscheint, schaltet sie die Fehlerverfolgung aus. Wenn die Anweisung innerhalb einer Fehlerbehandlungsroutine erscheint (wie das im nächsten Beispiel in der Routine `Behandlung` der Fall ist), veranlaßt sie BASIC dazu, dessen Standardmeldung für den vorliegenden Fehler auszugeben, und beendet das Programm.

Selbst dann, wenn das Programm eine Zeile mit der Zeilennummer 0 hat, signalisiert eine Anweisung **ON ERROR GOTO 0** daher dem Programm, die Fehlerverfolgung auszuschalten oder die Ausführung zu beenden.

6.1.2 Schreiben einer Fehlerbehandlungsroutine

Eine Fehlerbehandlungsroutine besteht aus drei Teilen:

1. der in der Anweisung **ON ERROR GOTO** *Zeile* angegebenen Zeilenmarke oder Zeilennummer, die die erste Anweisung darstellt, zu der das Programm nach einem Fehler verzweigt.

2. dem Hauptteil der Routine, der den die Verzweigung verursachenden Fehler identifiziert und auf jeden abgefangenen Fehler entsprechend reagiert.
3. mindestens einer **RESUME**- oder **RESUME NEXT**-Anweisung, um die Kontrolle aus dieser Routine zurückzugeben.

Eine Fehlerbehandlungsroutine sollte so plaziert werden, daß sie während des normalen Ablaufs der Ausführung nicht ausgeführt werden kann. Eine Fehlerbehandlungsroutine im Hauptmodul des Programms könnte beispielsweise vor einer **END**-Anweisung stehen. Andernfalls kann diese Routine möglicherweise nur anhand einer **GOTO**-Anweisung während der normalen Ausführung übersprungen werden.

6.1.2.1 Identifizieren von Fehlern mit Hilfe von ERR

Sobald ein Programm in eine Fehlerbehandlungsroutine verzweigt, muß es den Fehler bestimmen, der die Verzweigung verursacht hat. Die Fehlerquelle läßt sich anhand der Funktion **ERR** identifizieren. Diese Funktion gibt einen numerischen Code für den letzten Laufzeitfehler des Programmes zurück. (Eine komplette Liste der mit Laufzeitfehlern verknüpften Fehlercodes finden Sie in Tabelle I.1, "Aufruf-, Kompilierzeit- und Laufzeitfehlercodes".)

Hinweis Fehler können nicht innerhalb einer Fehlerbehandlungsroutine verfolgt werden. Wenn ein Fehler auftritt, während die Fehlerbehandlungsroutine einen anderen Fehler verarbeitet, gibt das Programm die Meldung für den neuen Fehler aus und beendet dann die Ausführung. Die Ereignisverfolgung wird damit auch unterbrochen, aber sie wird fortgesetzt, wenn QuickBASIC von der Fehlerbehandlungsroutine zurückkehrt. Das erste, nach Unterbrechung der Ereignisbehandlung auftretende Ereignis wird gespeichert.

Beispiel

Das folgende Programm verfügt über eine als `Behandlung` bezeichnete Fehlerbehandlungsroutine, die speziell zum Umgang mit drei unterschiedlichen Fehlersituationen bestimmt ist. Die in einer **SELECT CASE**-Anweisung verwendete **ERR**-Funktion ermöglicht es dieser Routine, auf jeden Fehler angemessen zu reagieren.

```
DATA BASIC, Pascal, FORTRAN, C, Modula, Forth
DATA LISP, Ada, COBOL
CONST FALSCH = 0, WAHR = NOT FALSCH
EndeDaten = FALSCH                ' Setze Ende des
                                  ' Datenkennzeichens.
ON ERROR GOTO Behandlung          ' Aktiviere Fehlerverfolgung.
OPEN "LPT1:" FOR OUTPUT AS #1     ' Öffne den Drucker für
                                  ' Ausgabe.
```

```
DO
   ' Fahre mit dem Lesen von Größen aus den obigen DATA-
   ' Anweisungen fort, gib sie dann auf den Zeilendrucker
   ' aus, bis eine Fehlermeldung "Außerhalb von DATA"
   ' anzeigt, daß keine weiteren Daten vorhanden sind:
   READ Puffer$
   IF NOT EndeDaten THEN
      PRINT #1, Puffer$
   ELSE
      EXIT DO
   END IF
LOOP

CLOSE #1
END

Behandlung:     ' Fehlerbehandlungsroutine

   ' Verwende ERR um zu bestimmen, welcher Fehler die
   ' Verzweigung in "Behandlung" verursacht hat:
   SELECT CASE ERR
      CASE 4

         ' 4 ist der Code für "Außerhalb von DATA" in
         ' DATA-Anweisungen:
         EndeDaten = WAHR
         RESUME NEXT
      CASE 25

         ' 25 ist der Code für "Gerätefehler", der durch den
         ' Versuch verursacht sein kann, Ausgabe auf den
         ' Drucker auszugeben, wenn dieser nicht
         ' eingeschaltet ist:
         PRINT "Schalten Sie den Drucker ein, betätigen ";
         PRINT "Sie dann eine beliebige Taste zum ";
         PRINT "Fortfahren."
         Pause$ = INPUT$(1)
         RESUME
      CASE 27

         ' 27 ist der Code für "Papier zu Ende":
         PRINT "Drucker hat kein Papier mehr. Legen ";
         PRINT "Sie Papier ein,betätigen Sie dann eine ";
         PRINT "beliebige Taste zum Fortfahren."
         Pause$ = INPUT$(1)
```

```
        ' Beginne, Daten vom Anfang der ersten DATA-
        ' Anweisung zu lesen, nachdem das Papier eingelegt
        ' ist:
        RESTORE
        RESUME
     CASE ELSE
        ' Ein vom Programmierer nicht abgefangener Fehler
        ' ist aufgetreten; zeige die Fehlermeldung für
        ' diesen Fehler an und beende das Programm:
        ON ERROR GOTO 0
  END SELECT
```

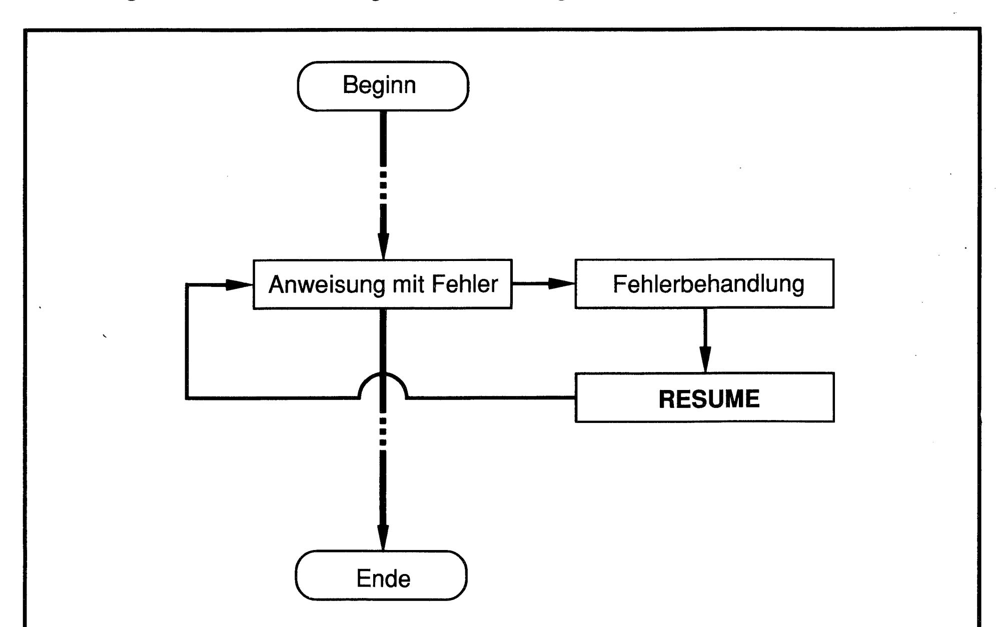
6.1.2.2 Verlassen einer Fehlerbehandlungsroutine

Die Anweisung **RESUME** gibt die Kontrolle von einer Fehlerbehandlungsroutine zurück. Das vorhergehende Beispiel verwendet folgende zwei Abwandlungen von **RESUME**, um die Routine `Behandlung` zu verlassen:

Anweisung	*Funktion*
RESUME	Veranlaßt das Programm, genau zu der Anweisung zurückzukehren, die den Fehler verursacht hat. Findet das Programm die aktive Fehlerbehandlungsroutine in einem anderen Modul, wird die Kontrolle an die letzte im jeweiligen Modul ausgeführte Anweisung zurückgegeben. Das vorhergehende Beispiel verwendet **RESUME**, um zu der **PRINT**-Anweisung zurückzukehren, die versucht hat, Ausgabe an den Drucker zu senden , und gibt der **PRINT**-Anweisung erneut die Möglichkeit, den Wert in `Puffer$` auszudrucken, nachdem der Drucker (vermutlich) eingeschaltet ist.
RESUME NEXT	Veranlaßt das Programm in die Anweisung zurückzuverzweigen, die der fehlerverursachenden Anweisung folgt. Findet das Programm die aktive Fehlerbehandlungsroutine in einem anderen Modul, wird die Kontrolle an die Anweisung zurückgegeben, die der letzten ausgeführten Anweisung in diesem Modul folgt. Das vorhergehende Programm verwendet die Anweisung **RESUME NEXT** zum Abfangen des Fehlers `Außerhalb von DATA`. Ein einfaches **RESUME** hätte in diesem Fall eine Endlosschleife zur Folge, da jedesmal, wenn die Kontrolle zu der **READ**-Anweisung im Hauptprogramm zurückkehrt, ein weiterer Fehler `Außerhalb von DATA` erscheint, der die Fehlerroutine ständig erneut aufruft.

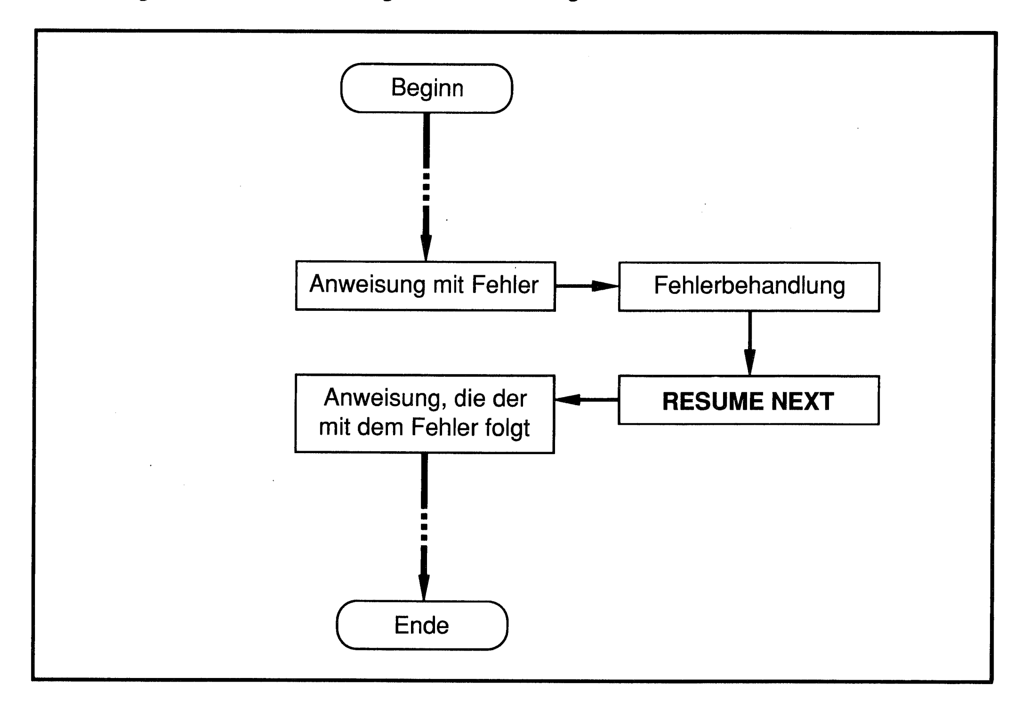
Abbildung 6.1 skizziert den Ablauf einer Programmsteuerung während einer Fehlerbehandlung mit **RESUME**:

Abbildung 6.1 Ablauf der Programmsteuerung mit RESUME



Vergleichen Sie die vorhergehende Abbildung mit Abbildung 6.2, die den Ablauf in einem Programm, das **RESUME NEXT** verwendet, darstellt:

Abbildung 6.2 Ablauf der Programmsteuerung mit RESUME NEXT



Eine weitere Variante von **RESUME** ist die Anweisung **RESUME** *Zeile*, wobei *Zeile* eine Zeilennummer oder Zeilenmarke außerhalb jeglicher **SUB...END SUB**-, **FUNCTION...END FUNCTION**- oder **DEF FN...END DEF**-Blöcke sein muß. Da sich *Zeile* außerhalb dieser Blöcke befinden muß, kann eine **RESUME** *Zeile*-Anweisung zu unerwarteten Nebenwirkungen führen, wenn sie in einer Fehlerbehandlungsroutine innerhalb einer **SUB**- oder **FUNCTION**-Prozedur oder einer **DEF FN**-Funktion erscheint. Sie sollten gewöhnlich zu Gunsten von **RESUME** oder **RESUME NEXT** vermieden werden.

6.2 Ereignisverfolgung

Abschnitt 6.2.1 stellt zwei Methoden vor, mit deren Hilfe BASIC-Programme Ereignisse entdecken können: Registrierung und Verfolgung. Die Abschnitte 6.2.2 bis 6.2.4 erläutern dann die verschiedenen Ereignisse, die BASIC verfolgen kann, wie eine Verfolgung gesetzt und aktiviert, sowie ausgesetzt oder deaktiviert wird. Die Verfolgung von Tastenbetätigungen und Musikereignissen wird in den Abschnitten 6.2.5 bis 6.2.7 im einzelnen beschrieben.

6.2.1 Erfassung der Ereignisse durch Registrierung

Eine Möglichkeit, das Auftreten eines Ereignisses zu erfassen und die Programmkontrolle in eine geeignete Routine zu leiten, bietet die Verwendung der "Ereignisregistrierung". Bei der Ereignisregistrierung unterbricht das Programm die laufende Ausführung und prüft ausdrücklich auf das Vorhandensein von Ereignissen. Die folgende Schleife prüft zum Beispiel die Tastatur bis der Benutzer entweder die B- oder die N-Taste betätigt:

```
DO
   Test$ = UCASE$(INKEY$)
LOOP UNTIL Test$ = "N" OR Test$ = "B"
```

Registrierung ist dann hilfreich, wenn die genaue Stelle im Programmablauf von vornherein bekannt ist, an der ein Ereignis geprüft werden soll.

6.2.2 Erfassung der Ereignisse durch Verfolgung

Nehmen Sie jedoch an, daß Sie das Programm nicht anhalten möchten oder daß Sie es zwischen jeder einzelnen Anweisung (das heißt, fortwährend) prüfen lassen möchten. In diesen Fällen ist die Registrierung schwerfällig oder unmöglich und die Verfolgung bietet eine bessere Möglichkeit.

Beispiel

Das folgende Beispiel stellt die Verwendung der Fehlerverfolgung dar:

```
' Veranlasse, daß das Programm immer dann in "FunktTast"
' verzweigt, wenn der Benutzer die Funktionstaste F1
' betätigt:
ON KEY(1) GOSUB FunktTast
```

```
' Schalte Verfolgung für die F1-Taste ein. BASIC prüft nun
' im Hintergrund auf dieses Ereignis bis zum Programmende,
' oder bis die Verfolgung durch KEY(1) OFF ausgeschaltet
' oder durch KEY(1) STOP ausgesetzt wird:
KEY(1) ON
OPEN "Daten" FOR INPUT AS #1
OPEN "Daten.Aus" FOR OUTPUT AS #2

DO UNTIL EOF(1)
   LINE INPUT #1, ZeilPuffer$
   PRINT #2, ZeilPuffer$
LOOP

FunktTast:    ' Jedesmal wenn der Benutzer F1 betätigt,
              ' verzweigt das Programm in diese Prozedur.
   .
   .
   .
RETURN
```

6.2.3 Angabe des zu verfolgenden Ereignisses und Aktivierung der Ereignisverfolgung

Wie aus dem vorhergehenden Beispiel ersichtlich, sind für die Ereignisverfolgung drei Schritte erforderlich:

1. Definieren einer Zielunterroutine, die als "Unterroutine zur Ereignisverfolgung" bezeichnet wird.
2. Dem Programm mit Hilfe einer **ON** *Ereignis* **GOSUB**-Anweisung mitteilen, in die Unterroutine zur Ereignisverfolgung zu verzweigen, wenn das Ereignis stattfindet.
3. Aktivieren der Verfolgung dieses Ereignisses mit Hilfe einer *Ereignis* **ON**-Anweisung.

Das Programm kann kein gegebenes Ereignis verfolgen, bis es nicht sowohl eine *Ereignis* **ON**- als auch eine **ON** *Ereignis* **GOSUB**-Anweisung angetroffen hat.

Eine Ereignisbehandlungsroutine ist eine normale BASIC **GOSUB**-Unterroutine: die Kontrolle geht zur ersten Zeile der Unterroutine über, sobald das Ereignis erfaßt wird, und kehrt mit Hilfe einer **RETURN**-Anweisung zurück auf die Hauptebene des Moduls, das die Unterroutine enthält.

Beachten Sie folgende wichtige Unterschiede zwischen der Syntax der Fehlerverfolgung und der Ereignisverfolgung:

Fehlerverfolgung	***Ereignisverfolgung***
Wird mit einer **ON ERROR GOTO**-Anweisung aktiviert, die auch die erste Zeile der Fehlerbehandlungsroutine angibt.	Wird mit einer *Ereignis* **ON**-Anweisung aktiviert. Eine separate **ON** *Ereignis* **GOSUB**-Anweisung gibt die erste Zeile der Unterroutine zur Ereignisbehandlung an.
Gibt die Kontrolle aus der Fehlerroutine mit Hilfe einer oder mehrerer RESUME-, RESUME NEXT- oder **RESUME** *Zeile*-Anweisungen zurück.	Gibt die Kontrolle aus der Ereignisunterroutine mit einer oder mehreren **RETURN**-Anweisungen zurück.

6.2.4 Von BASIC verfolgte Ereignisse

Innerhalb eines BASIC-Programmes können folgende Ereignisse verfolgt werden:

BASIC-Anweisung	***Verfolgtes Ereignis***
COM(*Anschlußnummer*)	Daten aus einem der seriellen Anschlüsse (1 oder 2), die im Datenübertragungspuffer erscheinen (der Zwischenspeicherbereich für Daten, die zum seriellen Anschluß gesendet oder von diesem empfangen werden).
KEY(*Tastennummer*)	Der Benutzer betätigt die angegebene Taste.
PEN	Der Benutzer aktiviert den Lichtstift.
PLAY(*Warteschlangengrenze*)	Die Anzahl der noch im Hintergrund zu spielenden Noten fällt unterhalb der *Warteschlangengrenze*.
STRIG(*Auslösenummer*)	Der Benutzer betätigt den Auslöser des Joysticks.
TIMER(*Intervall*)	Der Ablauf von *Intervall* Sekunden.

6.2.5 Aussetzen oder Ausschalten der Ereignisverfolgung

Sie können die Erfassung und Verfolgung jedes Ereignisses anhand der Anweisung *Ereignis* **OFF** ausschalten. Ereignisse, die nach der Ausführung einer *Ereignis* **OFF**-Anweisung auftreten, werden ignoriert. Wenn Sie die Verfolgung eines gegebenen Ereignisses aussetzen, das Ereignis aber weiterhin erfassen möchten, ist die Anweisung *Ereignis* **STOP** zu verwenden.

Nach *Ereignis* **STOP** verzweigt das Programm bei Auftreten des Ereignisses nicht in die Ereignisunterroutine. Das Programm behält jedoch, daß das Ereignis stattgefunden hat und nimmt nach Einschalten der Verfolgung mit *Ereignis* **ON** eine sofortige Verzweigung in die Ereignisunterroutine vor.

Unterroutinen zur Ereignisbehandlung führen eine implizite *Ereignis* **STOP**-Anweisung für ein gegebenes Ereignis immer dann aus, wenn sich die Programmkontrolle in der Routine befindet; dieser Anweisung folgt ein implizites *Ereignis* **ON** für dieses Ereignis, wenn die Programmkontrolle aus der Routine zurückgegeben wird. Wenn zum Beispiel eine Unterroutine zur Tastenbehandlung eine Tastenbetätigung bearbeitet, ist die Verfolgung dieser Taste bis zur vollständigen Bearbeitung der vorhergehenden Tastenbetätigung ausgesetzt. Wenn der Benutzer in der Zwischenzeit diese Taste betätigt, wird diese neue Tastenbetätigung festgehalten und verfolgt, nachdem die Kontrolle aus der Unterroutine zur Tastenbehandlung zurückgegeben wurde.

Beispiel

Eine Ereignisverfolgung unterbricht alle Vorgänge, die zu dem Zeitpunkt des Ereignisses im Programm ablaufen. Wenn Sie daher einen Block von Anweisungen nicht unterbrechen möchten, setzen Sie ein *Ereignis* **STOP** vor den Anweisungen und ein *Ereignis* **ON** nach den Anweisungen wie folgt ein:

```
' Verzweige einmal in der Minute (60 Sekunden) in die
' Unterroutine ZeigZeit:
ON TIMER (60) GOSUB ZeigZeit
' Aktiviere die Verfolgung des 60-Sekunden Ereignisses:
TIMER ON
.
.
.
TIMER STOP    'Verfolgung aussetzen
.
.        ' Eine Folge von Zeilen, die nicht unterbrochen
         ' werden sollen, selbst wenn mehr als 60 Sekunden
         ' vergangen sind
.
TIMER ON      'Verfolgung wieder aktivieren
.
.
.
END
ZeigZeit:
   ' Lies die aktuelle Zeilen- und Spaltenposition des
   ' Cursors und speichere diese in den Variablen Zeile und
   ' Spalte:
   Zeile  = CSRLIN
```

```
   Spalte = POS(0)
   ' Gehe in die 24-te Zeile, 20-te Spalte und gib die Zeit
   ' aus:
   LOCATE 24, 20
   PRINT TIME$
   ' Bringe den Cursor auf seine vorherige Position und
   ' kehre in das Hauptprogramm zurück:
   LOCATE Zeile, Spalte
RETURN
```

6.2.6 Verfolgung der Tastenbetätigungen

Um eine Tastenbetätigung zu erfassen und die Programmkontrolle in eine Unterroutine für Tastenbetätigung zu leiten, sind beide folgenden Anweisungen im Programm erforderlich:

ON KEY(*Tastennummer*) **GOSUB** *Zeile*
KEY(*Tastennummer*) **ON**

Der Wert *Tastennummer* entspricht hier den folgenden Tasten:

Wert	***Taste***
1-10	Funktionstasten F1-F10 (manchmal als "benutzerdefinierte Tasten" bezeichnet)
11	NACH OBEN-Richtungstaste (↑)
12	NACH LINKS-Richtungstaste (←)
13	NACH RECHTS-Richtungstaste (→)
14	NACH UNTEN-Richtungstaste (↓)
15-25	Benutzerdefinierte Tasten (siehe Abschnitte 6.2.6.1 bis 6.2.6.2
30	Die Funktionstaste F11 (nur bei 101-Tasten-Tastatur)
31	Die Funktionstaste F12 (nur bei 101-Tasten-Tastatur)

Beispiele

Folgende zwei Zeilen veranlassen das Programm bei Betätigung der Funktionstaste F2 in die Unterroutine `TastSub` zu verzweigen:

```
ON KEY(2) GOSUB TastSub
KEY(2) ON
.
.
.
```

Folgende vier Zeilen veranlassen das Programm bei Betätigung der Richtungstaste NACH UNTEN in die Unterroutine `UntenTaste` und bei Betätigung der Richtungstaste NACH OBEN in die Unterroutine `ObenTaste` zu verzweigen:

```
ON KEY(11) GOSUB ObenTaste
ON KEY(14) GOSUB UntenTaste
KEY(11) ON
KEY(14) ON
.
.
.
```

6.2.6.1 Verfolgung der benutzerdefinierten Tasten

Zusätzlich zur Unterstützung der vorbelegten Tastennummern 1-14 (plus 30 und 31 bei der 101-Tasten-Tastatur) ermöglicht BASIC es Ihnen, die Nummern 15-25 jeder der restlichen Tasten auf der Tastatur zuzuweisen. Die eigenen zu verfolgenden Tasten können anhand dieser drei Anweisungen definiert werden:

KEY *Tastennummer*, **CHR$(0)** + **CHR$**(*Abfragecode*)
ON KEY(*Tastennummer*) **GOSUB** *Zeile*
KEY(*Tastennummer*) **ON**

Hier ist *Tastennummer* ein Wert von 15 bis 25 und *Abfragecode* der Abfragecode für diese Taste. (Diese Codes sind in der ersten Spalte der Tabelle "Tastaturabfragecodes" in Anhang D zu finden.) Zum Beispiel veranlassen folgende Zeilen das Programm, bei Betätigung der T-Taste in die Unterroutine `TTaste` zu verzweigen:

```
' Definiere Taste 15 (der Abfragecode für "t" ist
' dezimal 20):
KEY 15, CHR$(0) + CHR$(20)
' Definiere die Verfolgung (Zielzeile, wenn "t" betätigt
' wird):
ON KEY(15) GOSUB TTaste
KEY(15) ON                ' Schalte Erfassung der Taste 15
                          ' ein.
PRINT "Betätigen Sie b zum Beenden."
DO                      ' Leere Schleife: warte, bis
LOOP UNTIL INKEY$ = "b"  ' Benutzer "b" betätigt.
END
TTaste:            ' Unterroutine zur Tastenbehandlung
   PRINT "t betätigt."
RETURN
```

6.2.6.2 Verfolgung der benutzerdefinierten umgeschalteten Tasten

Es können auch die "umgeschalteten" Tasten verfolgt werden. Eine Taste befindet sich im umgeschalteten Zustand, wenn sie gleichzeitig mit einer oder mehreren der Sondertasten UMSCHALTTASTE, STRG-TASTE oder ALT-TASTE, oder nach dem Einschalten der NUM-FESTSTELLTASTE oder UMSCHALT-FESTSTELLTASTE betätigt wird.

Nachfolgend wird die Verfolgung folgender Tastenkombinationen dargestellt:

KEY *Tastennummer*, **CHR$**(*Tastaturflag*) + **CHR$**(*Abfragecode*)
ON KEY(*Tastennummer*) **GOSUB** *Zeile*
KEY(*Tastennummer*) **ON**

An dieser Stelle ist *Tastennummer* ein Wert von 15 bis 25, *Abfragecode* ist der Abfragecode für die erste Taste und *Tastaturflag* ist die Summe der einzelnen Codes für die betätigten Sondertasten, wie in der folgenden Tabelle gezeigt:

Taste	***Tastencode***
UMSCHALTTASTE	1, 2 oder 3 Die Tastenverfolgung geht davon aus, daß die linke und rechte Umschalttaste identisch sind, so daß die UMSCHALTTASTE mit 1 (links), 2 (rechts) oder 3 (links + rechts) verfolgt werden kann.
STRG-TASTE	4
ALT-TASTE	8
NUM-FESTSTELLTASTE	32
UMSCHALT-FESTSTELLTASTE	64
Jede erweiterte Taste auf der 101-Tasten-Tastatur (mit anderen Worten, eine Taste wie die NACH LINKS -Richtungstaste oder ENTF-TASTE, die sich nicht auf dem Zehnerblock befindet)	128

Zum Beispiel schalten die folgenden Anweisungen die Verfolgung von STRG-TASTE + S ein. Beachten Sie die Gestaltung dieser Anweisungen, die die Verfolgung der Kombination STRG-TASTE + s (Kleinbuchstabe s) und STRG-TASTE + S (Großbuchstabe S) ermöglichen. Um den Großbuchstaben S zu verfolgen, muß das Programm, wie nachstehend gezeigt, sowohl durch das Niederdrücken der UMSCHALTTASTE als auch durch die Aktivierung der UMSCHALT-FESTSTELLTASTE erstellte Großbuchstaben erkennen:

```
' 31 = Tastaturabfragecode für die S-Taste
'  4 = Code für die STRG-Taste
KEY 15, CHR$(4) + CHR$(31)      ' Verfolge STRG+S.

' 5 = Code für STRG-Taste + Code für UMSCHALTTASTE
KEY 16, CHR$(5) + CHR$(31)      ' Verfolge STRG+UMSCHALT+S.

' 68 = Code für STRG-Taste + Code für UMSCHALT-
' FESTSTELLTASTE
KEY 17, CHR$(68) + CHR$(31)     ' Verfolge STRG+UMSCHALT-
                                ' FESTSTELL+S.

ON KEY (15) GOSUB StrgSVerfolg ' Teile Programm mit,  wohin
                                ' verzweigt werden soll.
ON KEY (16) GOSUB StrgSVerfolg ' (Beachte: dieselbe
                                ' Unterroutine für jede
                                ' Taste)
ON KEY (17) GOSUB StrgSVerfolg

KEY (15) ON                     ' Aktiviere Tastenerfassung
                                ' für alle drei
                                ' Kombinationen.
KEY (16) ON
KEY (17) ON
.
.
.
```

Die folgenden Anweisungen schalten die Verfolgung von STRG+ALT+ENTF ein:

```
' 12 = 4 + 8 = (Code für STRG-Taste) + (Code für ALT-Taste)
' 83 = Tastaturabfragecode für ENTF-Taste
KEY 20, CHR$(12) + CHR$(83)
ON KEY(20) GOSUB TastBehand
KEY (20) ON
.
.
.
```

Beachten Sie in dem vorhergehenden Beispiel, daß die BASIC-Ereignisverfolgung die normale Bedeutung von STRG+ALT+ENTF (Warmstart des Systems) aufhebt. Diese Verfolgung bietet eine einfache Möglichkeit, den Benutzer vor zufälligem Neustart der Maschine während eines Programmablaufes zu schützen.

Wenn Sie eine 101-Tasten-Tastatur verwenden, können Sie jede Taste auf dem erweiterten Tastaturblock durch Zuweisung der Zeichenkette

CHR$(128) + **CHR$(***Tastaturabfragecode*)

jedem Wert für *Tastennummer* von 15 bis 25, verfolgen.

Das nächste Beispiel zeigt, wie Sie die NACH LINKS-Richtungstaste sowohl auf dem erweiterten Cursor-Tastatur-Block als auch auf dem Zehnerblock verfolgen können.

```
' 128 = Tastaturflag für Tasten auf dem erweiterten Cursor-
' Tastaturblock
' 75 = Tastaturabfragecode für die NACH LINKS-Richtungstaste
KEY 15, CHR$(128) + CHR$(75) ' Verfolge NACH LINKS-Taste auf
                             ' dem erweiterten Cursor-
                             ' Tastaturblock.
ON KEY(15) GOSUB CursorBlock
KEY(15) ON
ON KEY(12) GOSUB ZehnerBlock ' Verfolge NACH LINKS-Taste auf
                             ' dem Zehnerblock.
KEY(12) ON
DO: LOOP UNTIL INKEY$ = "b"  ' Leere Schleife starten.
END
CursorBlock:
   PRINT " NACH LINKS-Taste wurde auf dem Cursorblock ";
   PRINT "betätigt."
RETURN
ZehnerBlock:
   PRINT " NACH LINKS-Taste wurde auf dem Zehnerblock ";
   PRINT "betätigt."
RETURN
```

Wichtig QuickBASIC übernimmt aus Kompatibilitätsgründen viele BASICA Konventionen, von denen eine die Verfolgung der FUNCTION-Tasten betrifft.

Wenn die Verfolgung der Funktionstasten(F1-F12) mit Hilfe der Anweisung **ON KEY**(*n*)**GOSUB** aufgerufen wird, verfolgen sowohl BASICA als auch QuickBASIC Fn, unabhängig davon, ob auch eine Umschalttaste (STRG, ALT, UMSCHALTTASTE) gleichzeitig betätigt wurde. Das bedeutet, daß die benutzerdefinierte Verfolgung für umgeschaltete Versionen dieser Funktionstaste ignoriert werden.

Wenn Sie daher eine Funktionstaste sowohl im umgeschalteten als auch im normalen Zustand verfolgen möchten, müssen Sie eine Benutzerdefinition für jeden Zustand erstellen. Nicht umschaltbare Tasten können weiterhin mit der Anweisung **ONKEY**(*n*)**GOSUB** verwendet werden.

6.2.7 Verfolgung von Musikereignissen

Die zum Spielen von Musik verwendete Anweisung **PLAY** bietet die Möglichkeit, die Musik im Vordergrund oder im Hintergrund spielen zu lassen. Wenn Sie die Standardeinstellung Vordergrundmusik wählen, kann bis zum Beenden der Musik kein anderer Vorgang ausgeführt werden. Wenn Sie jedoch die Option **MB** (Music Background) in einer **PLAY** Musikzeichenkette verwenden, spielt die Melodie im Hintergrund, während nachfolgende Anweisungen des Programmes ausgeführt werden.

Die **PLAY**-Anweisung spielt Musik im Hintergrund, indem sie den Puffer mit bis zu 32 Noten auffüllt, dann die Noten in dem Puffer spielt, während das Programm andere Vorgänge ausführt. Eine "Musik-Verfolgung" prüft fortwährend die Anzahl der im Puffer zum Spielen übriggebliebenen Noten. Sobald diese Anzahl unter die in der Verfolgung festgesetzten Grenze fällt, verzweigt das Programm in die erste Zeile der angegebenen Routine.

Zur Einrichtung einer Musik-Verfolgung im Programm, sind folgende Anweisungen erforderlich:

ON PLAY(*Grenze*)**GOSUB** *Zeile*
PLAY ON
PLAY "MB"
.
.
.
PLAY *Musikzeichenkette*
[**PLAY** *Musikzeichenkette*]
.
.
.

An dieser Stelle ist *Grenze* eine Zahl zwischen 1 und 32. Zum Beispiel veranlaßt dieser Ausschnitt das Programm in die Unterroutine `MusikVerfolg` zu verzweigen, sobald die Anzahl der im Musikpuffer verbliebenen Noten von acht auf sieben sinkt:

```
ON PLAY (8) GOSUB MusikVerfolg
PLAY ON
.
.
.
PLAY "MB"    ' Spiele nachfolgende Noten im Hintergrund.
PLAY "o1 A# B# C-"
.
.
.
```

```
MusikVerfolg:
   ' Unterroutine zur Musikverfolgung
RETURN
```

Wichtig Der Auslöser einer Musikverfolgung ist die von Grenze auf Grenze -1 sinkende Notenanzahl. Wenn zum Beispiel der Musikpuffer im vorhergehenden Beispiel nur sieben Noten enthält, findet die Verfolgung nicht statt. In dem Beispiel wird die Verfolgung nur aufgenommen, wenn die Notenanzahl von acht auf sieben sinkt.

Sie können anhand einer Unterroutine zur Musikverfolgung dasselbe Musikstück während der Programmausführung wiederholt spielen lassen, wie in dem nächsten Beispiel gezeigt:

```
' Schalte die Verfolgung der Hintergrundmusik-Ereignisse
' ein:
PLAY ON

' Verzweige zu der Unterroutine Auffrischen, wenn sich
' weniger als zwei Noten im Puffer der Hintergrundmusik
' befinden:
ON PLAY(2) GOSUB Auffrischen

PRINT "Mit beliebiger Taste starten, b zum Beenden."
Pause$ = INPUT$(1)

' Wähle für PLAY die Option Hintergrundmusik:
PLAY "MB"

' Beginne, die Musik zu spielen, so daß Noten in den Puffer
' der Hintergrundmusik geschrieben werden:
GOSUB Auffrischen

I = 0

DO

   ' Gib die Zahlen von 0 bis 10.000 solange aus, bis der
   ' Benutzer die "b"-Taste betätigt. Während dieses
   ' Verfahrens wird die Musik im Hintergrund wiederholt:
   PRINT I
   I = (I + 1) MOD 10001
LOOP UNTIL INKEY$ = "b"

END

Auffrischen:
   ' Spielt den Eröffnungssatz von Beethovens Fünfter
   ' Sinfonie:
   Beet$    = "t180 o2 p2 p8 L8 GGG L2 E-"
   Hoven$   = "p24 p8 L8 FFF L2 D"
   PLAY Beet$ + Hoven$
RETURN
```

6.3 Fehler- und Ereignisverfolgung in SUB- oder FUNCTION-Prozeduren

Bei der Verwendung von Fehler- oder Ereignisverfolgung mit BASIC-Prozeduren ist zu beachten, daß beide folgenden Anweisungen innerhalb eines **SUB...END SUB**- oder **FUNCTION...END FUNCTION**-Blockes zugelassen sind:

ON ERROR GOTO *Zeile*
ON *Ereignis* **GOSUB** *Zeile*

Die *Zeile*, auf die in der jeweiligen Anweisung Bezug genommen wird, muß jedoch eine Zeile im Modul-Ebenen-Code, nicht eine andere Zeile innerhalb der **SUB** oder **FUNCTION**, kennzeichnen.

Beispiel

Das folgende Beispiel zeigt die Plazierung einer Fehlerbehandlungsroutine, die innerhalb einer **SUB**-Prozedur verfolgten Fehler verarbeitet.

```
CALL KurzSub
END

AbfangFehler:
   ' Plaziere die Routine AbfangFehler auf der Modul-Ebene
   ' außerhalb des Unterprogrammes:
   PRINT "Fehler" ERR "abgefangen mit Fehlerbehandlung."
RESUME NEXT

SUB KurzSub STATIC
   ON ERROR GOTO AbfangFehler
   ERROR 62
END SUB
```

Ausgabe

```
Fehler 62, abgefangen mit Fehlerbehandlung.
```

6.4 Verfolgung in Mehrfachmodulen

Bis zu QuickBASIC 4.5 konnten nur Ereignisse in Modulen verfolgt werden. Sobald eine Ereignisbehandlungsroutine definiert und die Ereignisverfolgung in einem Modul aktiviert wurde, hat das Auftreten dieses Ereignisses während der Programmausführung in einem anderen Modul die Verzweigung in die Routine veranlaßt.

Es war nicht möglich Fehler in Modulen zu verfolgen. Trat ein Fehler in einem Modul auf, das nicht über eine aktive Fehlerbehandlung verfügte, wurde die Programmausführung beendet, obwohl eine Fehlerbehandlung, die das Problem lösen konnte, in einem anderen Modul vorhanden war.

QuickBASIC erweitert die Reichweite der Fehlerverfolgung. Es ist jetzt möglich, sowohl Fehler als auch Ereignisse in Modulen zu verfolgen. Die nächsten beiden Abschnitte erklären diesen Vorgang und die geringfügigen noch bestehenden Unterschiede zwischen Ereignis- und Fehlerverfolgung.

6.4.1 Ereignisverfolgung innerhalb von mehreren Modulen

Die Ausgabe des folgenden Programmes zeigt, daß eine in dem Hauptmodul für die Funktionstaste F1 eingesetzte Verfolgung sogar dann ausgelöst wird, wenn sich die Programmkontrolle in einem anderen Modul befindet:

```
' ==========================================================
'                      MODUL
' ==========================================================
ON KEY (1) GOSUB F1Taste
KEY (1) ON
PRINT "Im Hauptmodul. Betätigen Sie f zum Fortfahren."
DO : LOOP UNTIL INKEY$ = "f"

CALL SubTaste

PRINT "Zurück im Hauptmodul. Betätigen Sie b zum Beenden."
DO : LOOP UNTIL INKEY$ = "b"
END

F1Taste:
  PRINT "Bearbeitung der F1-Tastenbetätigung im Hauptmodul."
RETURN
```

```
' ==========================================================
'                         MODUL SUBTASTE
' ==========================================================
SUB SubTaste STATIC
   PRINT "Im Modul mit SUBTASTE. Betätigen Sie zum ";
   PRINT "zurückzukehren."
   ' Die Betätigung von F1 an dieser Stelle ruft immer noch
   ' die Unterroutine F1Taste im Hauptmodul auf:
   DO : LOOP UNTIL INKEY$ = "z"
END SUB
```

Ausgabe

```
Im Hauptmodul. Betätigen Sie f zum Fortfahren.
Bearbeitung der F1-Tastenbetätigung im Hauptmodul.
Im Modul mit SUBTASTE. Betätigen Sie z, um zurückzukehren.
Bearbeitung der F1-Tastenbetätigung im Hauptmodul.
Zurück im Hauptmodul. Betätigen Sie b zum Beenden.
Bearbeitung der F1-Tastenbetätigung im Hauptmodul.
```

6.4.2 Fehlerverfolgung innerhalb von mehreren Modulen

Fehler können auch innerhalb von Mehrfachmodulen verfolgt werden. Verfügt das Modul, in dem der Fehler aufgetreten ist, nicht über eine aktive Fehlerbehandlung, sucht BASIC nach ihr in dem Modul, aus dem das aktive Modul aufgerufen wurde, und geht dabei soweit zurück, bis es ein Modul mit einer aktiven Fehlerbehandlung findet. Wenn das nicht der Fall ist, erscheint eine Fehlermeldung und die Programmausführung wird angehalten.

Es ist zu beachten, daß BASIC nicht alle Module durchsucht, sondern nur die, die sich auf dem Aufrufpfad zum fehlerverursachenden Code befinden.

Eine Anweisung **ON ERROR** muß vor Auftreten eines Fehlers ausgeführt werden, so daß BASIC nach einer Behandlung suchen kann. Deshalb ist die Anweisung **ON ERROR GOTO** im Ablauf des Programmflusses so einzusetzen, daß die Ablaufkontrolle diese erreichen kann. Diese Anweisung kann beispielsweise innerhalb einer vom Hauptmodul aufgerufenen Prozedur plaziert werden (siehe nachstehende Beispiele).

Beispiel

Das folgende Beispiel zeigt, wie Fehler für eine Prozedur in einer Quick-Bibliothek verfolgt werden. Die Prozedur `AdapterTyp` in dieser Bibliothek prüft mit allen möglichen **SCREEN**-Anweisungen die von der Computerhardware unterstützten graphischen Bildschirmmodi. (Weitere Informationen zu Quick-Bibliotheken finden Sie in Anhang H, "Erstellung und Verwendung von Quick-Bibliotheken".)

```
' ========== MODUL-EBENEN-CODE IN QUICK-BIBLIOTHEK==========
' Die Fehlerbehandlungsroutinen für Prozeduren in dieser
' Bibliothek müssen auf dieser Ebene definiert sein. Diese
' Ebene wird nur ausgeführt, wenn in den Prozeduren ein
' Fehler auftritt.
' =========================================================
DEFINT A-Z
CONST FALSCH = 0, WAHR = NOT FALSCH
.
.
.
CALL AdapterTyp
.
.
.
END
RetteMich:       ' Fehlerbehandlungsroutine
   AnzeigeFehler = WAHR
   RESUME NEXT
' ======== PROZEDUR-EBENEN-CODE IN QUICK-BIBLIOTHEK ========
'   Fehlerverfolgung wird auf dieser Ebene aktiviert.
' =========================================================
SUB AdapterTyp STATIC
   SHARED AnzeigeFehler   ' Die Variable AnzeigeFehler wird
                          ' gemeinsam mit der Fehlerroutine
                          ' RetteMich im obigen Modul-
                          ' Ebenen-Code benutzt.
   DIM MonitorTyp(1 TO 13)' Dimensioniere Datenfeld
                          ' MonitorTyp.
   J = 1                  ' Initialisiere Indexzähler für
                          ' Datenfeld MonitorTyp.
ON ERROR GOTO RetteMich         ' Schalte Fehlerverfolgung
                                ' ein.
```

```
   FOR Test = 13 TO 1 STEP -1
      SCREEN Test                 ' Versuche eine SCREEN-
                                  ' Anweisung um
                                  ' festzustellen, ob sie
                                  ' einen Fehler verursacht.
      IF NOT AnzeigeFehler THEN   ' Kein Fehler, dies ist
                                  ' also ein gültiger
                                  ' Bildschirmmodus.
         MonitorTyp(J) = Test     ' Speichere den Modus in
                                  ' dem Datenfeld.
         J = J + 1                ' Zähle den Indexzähler
                                  ' hoch.
      ELSE
         AnzeigeFehler = FALSCH   ' Fehler, also setze
                                  ' AnzeigeFehler zurück.
      END IF
   NEXT Test

   SCREEN 0, 0        ' Setze 80-Spalten Textmodus.
   WIDTH 80
   LOCATE 5, 10
   PRINT "Ihr Computer unterstützt die folgenden ";
   PRINT "Bildschirmmodi":
   PRINT
   FOR I = 1 TO J      ' Gib die Modi aus, die keine Fehler
                       ' verursachen.
      PRINT TAB(20); "BILDSCHIRM"; MonitorTyp(I)
   NEXT I

   LOCATE 20, 10
   PRINT "Weiter mit beliebiger Taste..."

   DO: LOOP WHILE INKEY$ = ""
END SUB
```

Wenn die Fehlerbehandlungsroutine in jedem Modul genau dasselbe Verfahren ausführen soll, kann jede Routine, wie im nächsten Beispiel gezeigt, zum Aufruf einer gemeinsamen fehlerverarbeitenden **SUB**-Prozedur veranlaßt werden:

```
' ==========================================================
'                          HAUPTMODUL
' ==========================================================

DECLARE SUB GlobalBehandlung (ModulName$)
DECLARE SUB KurzSub ()
```

```
ON ERROR GOTO LokalBehandlung
ERROR 57                ' Simuliere Auftreten des Fehlers 57
                        ' ("Geräte-E/A-Fehler").
KurzSub                 ' Rufe die SUB KurzSub auf.
END

LokalBehandlung:
   ModulName$ = "HAUPT"
   CALL GlobalBehandlung(ModulName$) ' Rufe SUB
GlobalBehandlung auf.

RESUME NEXT
' ==========================================================
'                       MODUL KURZSUB
' ==========================================================
DECLARE SUB GlobalBehandlung (ModulName$)

LokalBehandlung:
   ModulName$ = "KURZSUB"
   GlobalBehandlung ModulName$ ' Rufe GlobalBehandlung auf.
RESUME NEXT

SUB KurzSub STATIC
   ON ERROR GOTO LokalBehandlung
   ERROR 13             ' Simuliere den Fehler "Unverträgliche
                        ' Datentypen".
END SUB
' ==========================================================
'                  MODUL GLOBALBEHANDLUNG
' ==========================================================

SUB GlobalBehandlung (ModulName$) STATIC
   PRINT "Verfolgter Fehler";ERR;"in";ModulName$;"Modul"
END SUB
```

Ausgabe

```
Verfolgter Fehler 57 im Modul HAUPT.
Verfolgter Fehler 13 im Modul KURZSUB.
```

6.5 Fehler- und Ereignisverfolgung in Programmen, die mit BC kompiliert sind

Wenn beide der folgenden Aussagen für das Programm zutreffen, müssen Sie zum Kompilieren von Programmen die entsprechende untenaufgeführte BC-Befehlszeilenoption verwenden:

- Sie entwickeln das Programm außerhalb der QuickBASIC-Umgebung, das bedeutet, daß Sie den BASIC-Quellcode anhand eines anderen Texteditors schreiben und anschließend mit Hilfe der Befehle BC und LINK ein selbständig lauffähiges Programm aus diesem Quellcode erzeugen.
- Ihr Programm enthält eine der nachfolgenden in der zweiten Spalte aufgeführten Anweisungen.

Tabelle 6.1 BC-Befehlszeilenoptionen für Fehler- und Ereignisverfolgung

Option der Befehlszeile	***Anweisungen***
/E	**ON ERROR GOTO RESUME** *Zeile*
	Die Option /E zeigt dem Compiler an, daß Ihr Programm seine eigene Fehlerverfolgung mit **ON ERROR GOTO**- und **RESUME**-Anweisungen durchführt.
/V	**ON** *Ereignis* **GOSUB** *Ereignis* **ON**
	Die Option /V weist das Programm an, das Auftreten des angegebenen *Ereignisses* zwischen jeder Anweisung zu prüfen (diese Auswirkung ist mit der von /W zu vergleichen.)
/W	**ON** *Ereignis* **GOSUB** *Ereignis* **ON**
	Die Option /W zeigt dem Programm an, das Auftreten des angegebenen *Ereignisses* zwischen jeder Zeile zu prüfen. (Diese Auswirkung ist mit der von /V zu vergleichen.)
	Da BASIC mehrere Anweisungen auf einer Zeile zuläßt, prüfen die mit /W kompilierten Programme nicht so häufig wie die mit /V kompilierten Programme.
/X	**RESUME RESUME NEXT RESUME 0**
	Die Option /X zeigt dem Compiler an, daß im Programm eine der obigen Formen von **RESUME** verwendet wurde, um die Kontrolle aus einer Fehlerbehandlungsroutine zurückzugeben.

Beispiel

Die folgenden DOS-Befehlszeilen kompilieren und binden ein Modul mit dem Namen *rmtab.bas* und erzeugen ein selbständiges Programm mit dem Namenn *rmtab.exe*. (Da dieses Programm ohne die Option /O kompiliert ist, erfordert es die Laufzeit-Bibliothek *brun40.lib*, um lauffähig zu sein. Weitere Informationen zum Kompilieren und Binden finden Sie im Anhang G, "Kompilieren und Binden aus DOS".) Da dieses Modul **ON ERROR GOTO**- und **RESUME NEXT**-Anweisungen enthält, sind für das Kompilieren die Optionen /E und /X erforderlich.

```
BC RMTAB , , /E /X;
LINK RMTAB;
```

6.6 Beispielanwendung: Verfolgen von Dateizugriffsfehlern (*datfehl.bas*)

Das folgende Programm erhält als Eingabe sowohl einen Dateinamen als auch eine Zeichenkette, die in der Datei gesucht werden soll. Es listet dann alle Zeilen der gegebenen Datei auf, die die angegebene Zeichenkette enthalten. Wenn ein Dateizugriffsfehler auftritt, wird er mit der Routine `FehlerProz` verfolgt und bearbeitet.

Verwendete Anweisungen und Funktionen

Dieses Programm veranschaulicht die Verwendung der folgenden Anweisungen und Funktionen zur Fehlerbehandlung:

- **ERR**
- **ON ERROR GOTO**
- **RESUME**
- **RESUME NEXT**

Programm-Listing

```
' Deklariere die im Programm verwendeten symbolischen
' Konstanten:
CONST FALSCH = 0, WAHR = NOT FALSCH
DECLARE FUNCTION LiesDateiName$ ()
' Schalte die FEHLER-Verfolgung ein und gib den Namen
' der Fehlerbehandlungsroutine an:
```

```
ON ERROR GOTO FehlerProz
DO
   Neustart = FALSCH
   CLS
   DateiName$ = LiesDateiName$    ' Eingabe des
                                  ' Dateinamens.
   IF DateiName$ = "" THEN
      END                         ' Beende, wenn die
                                  ' <EINGABETASTE>
                                  ' betätigt wird.
   ELSE
      ' Anderenfalls öffne die Datei; weise ihr dabei
      ' die nächste verfügbare Dateinummer zu
      DateiNum = FREEFILE
      OPEN DateiName$ FOR INPUT AS DateiNum
   END IF
   IF NOT Neustart THEN
      ' Gib zu suchende Zeichenkette ein:
      PRINT "Geben Sie die zu suchende Zeichenkette ";
      LINE INPUT "ein: ", SuchKette$
      SuchKette$ = UCASE$(SuchKette$)
      ' Durchlaufe die Zeilen der Datei und gib diese
      ' aus, wenn sie die zu suchende Zeichenkette
      ' enthalten:
      ZeileNum = 1
      DO WHILE NOT EOF(DateiNum)
         ' Einlesen einer Zeile aus der Datei:
         LINE INPUT #DateiNum, ZeilePuffer$
         ' Prüfe auf Zeichenkette und gib die Zeile
         ' sowie deren Nummer aus, wenn gefunden:
         IF INSTR(UCASE$(ZeilePuffer$), SuchKette$) <> 0_
           THEN
            PRINT USING "#### &"; ZeileNum, ZeilePuffer$
         END IF
         ZeileNum = ZeileNum + 1
      LOOP
      CLOSE DateiNum       ' Schließe die Datei.
   END IF
LOOP WHILE Neustart = WAHR
```

```
END
FehlerProz:
   SELECT CASE ERR
      CASE 64:               ' Unzulässiger Dateiname
         PRINT "** FEHLER - Unzulässiger Dateiname"
         ' Lies einen neuen Dateinamen und versuche
         ' erneut:
         DateiName$ = LiesDateiName$
         ' Fahre mit der Anweisung fort, die den
         ' Fehler verursachte:
         RESUME
      CASE 71:                ' Diskette nicht bereit
         PRINT "** FEHLER - Diskette nicht bereit"
         PRINT "Betätige F zum Fortfahren, N zum ";
         PRINT "Neustart, B zum Beenden: "
         DO
           Zeich$ = UCASE$(INPUT$(1))
           IF Zeich$ = "F" THEN
              RESUME         ' Fahre dort fort, wo
                             ' beendet wurde.
           ELSEIF Zeich$ = "N" THEN
              Neustart = WAHR   ' Fahre am Beginn
              RESUME NEXT       ' fort.
           ELSEIF Zeich$ = "B" THEN
              END        ' Fahre überhaupt nicht fort.
           END IF
         LOOP
      CASE 53, 76:              ' Datei oder Pfad
                                ' nicht gefunden.
         PRINT "** FEHLER - Datei oder Pfad nicht gefunden"
         DateiName$ = LiesDateiName$
         RESUME
      CASE ELSE:              ' Unvorhergesehener Fehler
         ' Schalte Fehlerverfolgung aus und gib
         ' Standard-Systemmeldung aus:
         ON ERROR GOTO 0
   END SELECT
```

```
'
' ================== LIESDATEINAME$ ======================
'   Gibt eine vom Benutzer eingegebene Datei zurück
' ==========================================================
'
FUNCTION LiesDateiName$ STATIC
   PRINT "Gib zu durchsuchende Datei ein ";
   PRINT "(betätige EINGABETASTE zum Beenden): "
   INPUT DTemp$
   LiesDateiName$ = DTemp$
END FUNCTION
```

7 Programmieren mit Modulen

Dieses Kapitel zeigt, wie Sie Ihre Programmierprojekte durch Einteilung in "Module" überschaubar gestalten können. Module weisen eine leistungsfähige Organisationsfunktion auf, da sie die Einteilung eines Programmes in logisch zusammenhängende Teile erlauben, anstatt alle Codes in einer Datei zu behalten.

Dieses Kapitel stellt die Verwendung von Modulen zu folgenden Zwecken vor:

- Schreiben und Prüfen von neuen Prozeduren getrennt vom Rest des Programmes.
- Erstellen der eigenen **SUB**- und **FUNCTION**-Prozedurbibliotheken, die jedem neuen Programm hinzugefügt werden können.
- Kombinieren von Routinen anderer Sprachen (wie C oder MASM) mit den BASIC-Programmen.

7.1 Zweckmäßigkeit der Module

Ein Modul ist eine Datei, die einen ausführbaren Programmteil enthält. Ein vollständiges Programm kann in einem einzigen Modul enthalten oder auf zwei oder mehrere Module aufgeteilt sein.

Bei der Aufteilung eines Programmes in Module werden logisch zusammenhängende Abschnitte in separate Dateien geschrieben. Diese Organisation kann das Schreiben, Prüfen und die Fehlerbeseitigung beschleunigen und vereinfachen.

Die Aufteilung eines Programms in Module weist folgende Vorteile auf:

- Module erlauben das getrennte Schreiben von Prozeduren und deren nachträgliche Kombination mit dem Rest des Programmes. Dieses Verfahren ist besonders hilfreich beim Prüfen der Prozeduren, die somit außerhalb der Programmumgebung überprüft werden können.
- Zwei oder mehrere Programmierer haben die Möglichkeit an unterschiedlichen Teilen desselben Programmes zu arbeiten, ohne dauernd in Verbindung zu stehen. Dies ist besonders hilfreich, um komplexe Programmierprojekte zu bearbeiten.

- Sie können Ihren eigenen Programmierbedürfnissen entsprechende Prozeduren in einem eigenen Modul erstellen und in neuen Programmen einfach durch Laden des jeweiligen Moduls wiederverwenden.
- Mehrfachmodule vereinfachen die Softwarewartung. Eine von mehreren Programmen benutzte Prozedur kann sich in einem Modul der Bibliothek befinden und es genügt, eventuell erforderliche Änderungen nur einmal vorzunehmen.

7.2 Hauptmodule

Das Modul, das die erste ausführbare Anweisung des Programmes enhält, wird als "Hauptmodul" bezeichnet. Diese Anweisung gehört nie zu einer Prozedur, da die Ausführung nicht mit einer Prozedur beginnen kann.

Der gesamte Inhalt eines Moduls, mit Ausnahme der **SUB**- und **FUNCTION**-Prozeduren, befindet sich auf dem sogenannten "Modul-Ebenen-Code". In QuickBASIC bezeichnet der Modul-Ebenen-Code alle Anweisungen, auf die der Zugriff ohne Umschaltung auf ein prozeduredítierendes Fenster möglich ist. Abbildung 7.1 veranschaulicht den Zusammenhang zwischen diesen Elementen.

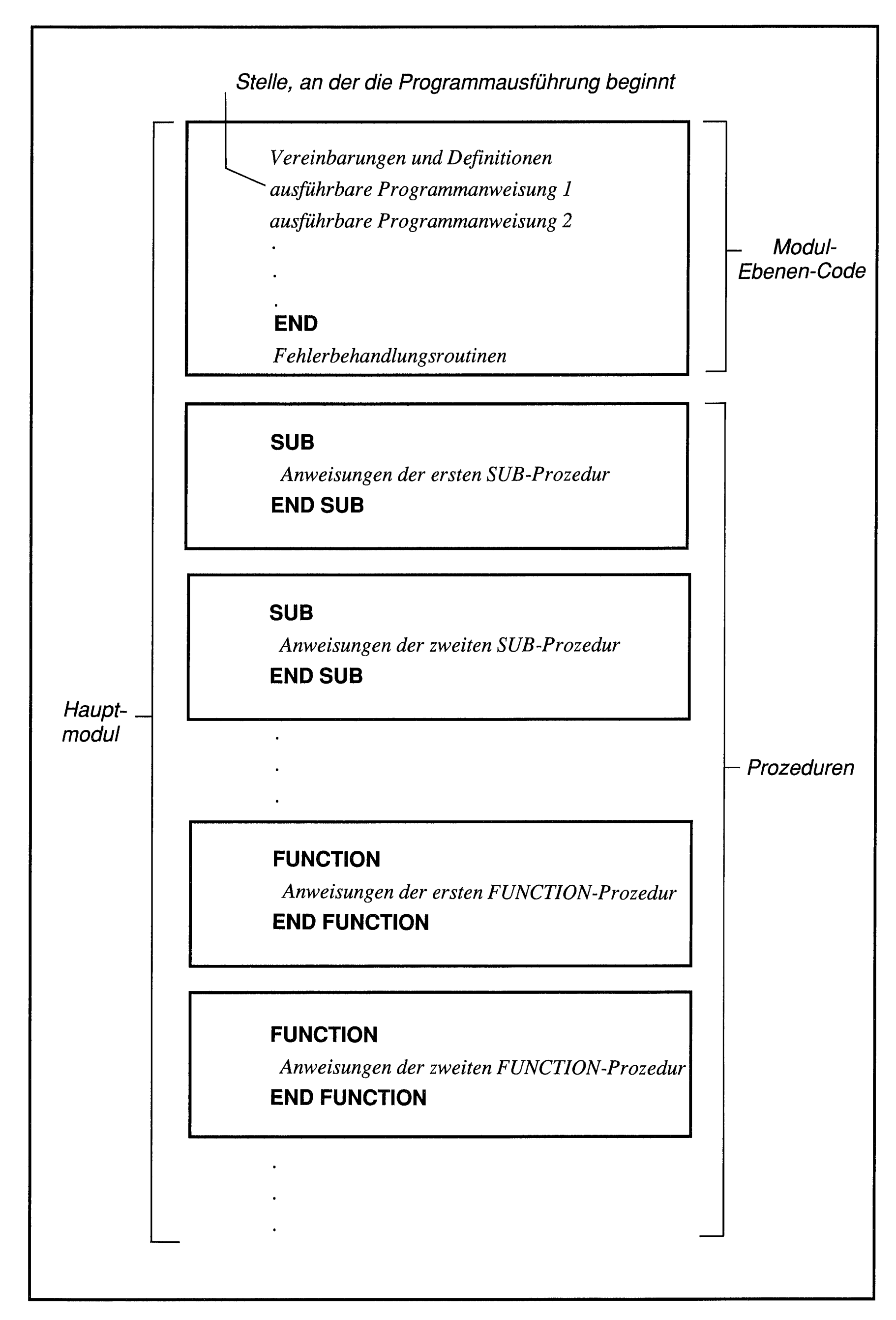
7.3 Module, die nur Prozeduren enthalten

Ein Modul braucht keinen Modul-Ebenen-Code zu enthalten, es kann lediglich aus **SUB**- und **FUNCTION**-Prozeduren bestehen, die die wichtigste Anwendung der Module darstellen.

Module werden oft zum "Trennen" der Prozeduren vom Hauptteil des Programmes verwendet. Dadurch kann beispielsweise ein Projekt unter mehreren Programmierern aufgeteilt werden. Wenn Mehrzweckprozeduren erstellt werden, die in zahlreichen Programmen zur Anwendung kommen (wie zum Beispiel Prozeduren, die Matrizen auswerten, Binärdaten zu einem **COM**-Anschluß senden, Zeichenketten verändern oder Fehler behandeln), können diese ebenfalls in Modulen gespeichert und anschließend in neuen Programmen einfach durch das Laden des entsprechenden Moduls in QuickBASIC wiederverwendet werden.

Hinweis Benötigt eine Prozedur in einem ausschließlich aus Prozeduren bestehenden Modul eine Fehler- oder Ereignisbehandlungsroutine oder eine Anweisung **COMMON SHARED**, werden diese auf Modul-Ebene eingesetzt.

Abbildung 7.1 Hauptmodul mit Modul-Ebenen-Codes und Prozeduren



7.4 Erstellen eines ausschließlich aus Prozeduren bestehenden Moduls

Ein ausschließlich Prozeduren enthaltendes Modul ist leicht zu erstellen. Neue Prozeduren können in einer separaten Datei eingegeben und von einer Datei in die andere übertragen werden.

Ein neues Modul läßt sich wie folgt erstellen:

1. Rufen Sie QuickBASIC auf, ohne Dateien zu öffnen oder zu laden.
2. Schreiben Sie alle gewünschten **SUB**- und **FUNCTION**-Prozeduren, ohne jedoch einen Modul-Ebenen-Code einzugeben. (Routinen zur Fehler- oder Ereignisverfolgung, sowie benötigte BASIC-Vereinbarungen bilden dabei die Ausnahme.)
3. Speichern und benennen Sie dieses Modul anhand des **Speichern Als-**Befehls.

Prozeduren lassen sich wie folgt von einem Modul in das andere übertragen:

1. Laden Sie die Dateien, die die zu übertragenden Prozeduren enthalten.
2. Wenn die Zieldatei bereits existiert, laden Sie diese aus dem Dateimenü ebenfalls anhand des Befehls **Datei laden**. Ist diese noch nicht vorhanden, legen Sie die neue Datei mit Hilfe der Option **Datei erstellen** aus dem Dateimenü an.
3. Wählen Sie SUBs aus dem Ansichtsmenü und übertragen Sie die Prozeduren von der alten in die neue Datei mit Hilfe des Befehls **Bewegen**. Diese Übertragung wird endgültig, wenn Sie QuickBASIC verlassen und die Frage des Dialogfeldes, ob die geänderten Dateien abzuspeichern sind, mit Ja beantworten. Andernfalls bleiben die Prozeduren an ihrer ursprünglichen Stelle.

7.5 Laden der Module

In QuickBASIC läßt sich eine beliebige (nur vom verfügbaren Speicherplatz begrenzte) Anzahl von Modulen mit Hilfe des Befehls **Datei laden** aus dem Dateimenü laden. Sämtliche Prozeduren der geladenen Module können von jeder anderen Prozedur oder vom Modul-Ebenen-Code aufgerufen werden. Es ist belanglos, ob ein Modul zufällig eine Prozedur enthält, die nie aufgerufen wird.

Sämtliche geladene Module können Modul-Ebenen-Code enthalten. Normalerweise beginnt QuickBASIC die Ausführung mit dem Modul-Ebenen-Code des ersten geladenen Moduls. Soll die Ausführung in einem anderen Modul beginnen, ist der Befehl **Hauptmodul festlegen** aus dem Menü **Ausführen** zu verwenden. Die Ausführung wird normalerweise am Ende des festgelegten Hauptmoduls angehalten; auf Grund seines Aufbaus setzt QuickBASIC die Ausführung nicht mit dem Modul-Ebenen-Code anderer Module fort.

Die Fähigkeit den auszuführenden Modul-Ebenen-Code auszuwählen ist beim Vergleich zweier Versionen desselben Programms behilflich. Es kann zum Beispiel vorkommen, daß Sie die unterschiedlichen Benutzerschnittstellen durch Einsatz in getrennten Modulen prüfen möchten. Sie können auch Testcode in einem ausschließlich aus Prozeduren bestehenden Modul plazieren und anschließend den Befehl **Hauptmodul festlegen** zum Wechsel zwischen dem Programm und den Tests verwenden.

Die vom Programm benutzten Module brauchen nicht erfaßt zu werden. Wenn Sie den Befehl **Alles speichern** verwenden, erstellt (oder aktualisiert) QuickBASIC eine *.mak* Datei, die alle gegenwärtig geladenen Module auflistet. Beim nächsten Laden des Hauptmoduls anhand des Befehls **Programm laden**, sieht QuickBASIC in der *.mak*-Datei nach und lädt automatisch die darin aufgelisteten Module.

7.6 Verwendung der DECLARE-Anweisung mit Mehrfachmodulen

Die Anweisung **DECLARE** hat folgende wichtige Funktionen in QuickBASIC:

1. Angabe der Sequenz und der Datentypen von Prozedurparametern.
2. Gewährleistung der Typenüberprüfung, die beim Aufruf jeder Prozedur die Übereinstimmung der Argumente mit den Parametern in Anzahl und Datentyp bestätigt.
3. Identifizierung eines **FUNCTION**-Prozedurnamens als Prozedur- nicht als Variablenname.
4. Das Wichtigste ist jedoch, es dem Hauptmodul zu ermöglichen, Prozeduren aus anderen Modulen (oder Quick-Bibliotheken) aufzurufen.

QuickBASIC verfügt über ein eigenes System für das automatische Einfügen der erforderlichen **DECLARE**-Anweisungen in Modulen. Abschnitt 2.5.4, "Überprüfen von Argumenten anhand der DECLARE-Anweisung" erläutert die Möglichkeiten und Einschränkungen dieses Systems.

Trotz des automatischen Einfügens von **DECLARE**-Anweisungen, das in QuickBASIC zur Verfügung steht, kann es vorkommen, daß Sie eine getrennte Include-Datei erstellen möchten, die alle für das Programm erforderlichen **DECLARE**-Anweisungen enthält. Diese Datei kann beim Einfügen und Löschen von Prozeduren oder bei der Änderung der Argumentenlisten manuell aktualisiert werden.

Wenn Sie Programme anhand eines Texteditors schreiben (anstatt der QuickBASIC Programmierumgebung) und diese mit BC kompilieren, müssen die **DECLARE**-Anweisungen manuell hinzugefügt werden.

7.7 Variablenzugriff von zwei oder mehreren Modulen

Mit Hilfe des Attributes **SHARED** erhalten Sie Zugriff auf die Variablen der Modul-Ebene und auf die der Prozeduren innerhalb des Moduls. Bei der Übertragung dieser Prozeduren in ein anderes Modul werden diese Variablen jedoch nicht mehr gemeinsam benutzt.

Diese Variablen können jeder Prozedur durch deren Argumentenliste übergeben werden. Diese Möglichkeit ist aber ziemlich unbequem, wenn es um die Übergabe zahlreicher Variablen geht.

Eine Lösung ist die Verwendung von **COMMON**-Anweisungen, die zwei oder mehreren Modulen die Möglichkeit bieten, auf dieselbe Variablengruppe zuzugreifen. Abschnitt 2.6, "Die gemeinsame Nutzung von Variablen anhand von SHARED", beschreibt dieses Verfahren im einzelnen.

Eine andere Lösung besteht in der Verwendung der Anweisung **TYPE...END TYPE** zur Kombination sämtlicher zu übergebenden Variablen in einer einzigen Struktur. Die Argument- und Parameterlisten brauchen dann unabhängig von der Anzahl der übergebenen Variablen nur einen Variablennamen zu enthalten.

Werden ein Programm und die dazugehörigen Prozeduren einfach in separate Module aufgeteilt, sind beide Lösungen einsatzfähig. Wenn, andererseits, eine Prozedur einem Modul (zur Verwendung in anderen Programmen) hinzugefügt wird, sollte die Anweisung **COMMON** vermieden werden. Module sind dazu bestimmt, das Ergänzen neuer Programme mit existierenden Prozeduren zu vereinfachen und in diesem Fall würden die **COMMON**-Anweisungen den Vorgang komplizieren. Falls eine Prozedur eine größere Variablengruppe beansprucht, gehört sie wahrscheinlich nicht in ein separates Modul.

7.8 Verwendung der Module während der Programmentwicklung

Am Anfang eines neuen Programmierprojektes ist es empfehlenswert, in den bereits vorhandenen Modulen nach Prozeduren zu suchen, die in der neuen Software verwertet werden können. Befinden sich die betreffenden Prozeduren nicht bereits in einem separaten Modul, sollte deren Übertragung in Erwägung gezogen werden.

Während der Programmerstellung werden neu geschriebene Prozeduren automatisch in die Programmdatei (das heißt, in das Hauptmodul) eingegliedert. Diese Prozeduren können für Prüfzwecke in ein separates Modul übertragen, oder einem Ihrer eigenen Module mit anderen in verschiedenen Programmen verwendeten Mehrzweckprozeduren hinzugefügt werden.

Es kann sein, daß Ihr Programm in anderen Sprachen geschriebene Prozeduren erfordert. (MASM ist zum Beispiel ideal als direkte Schnittstelle mit der Hardware, FORTRAN verfügt über fast jede erdenkliche mathematische Funktion, Pascal ermöglicht die Erstellung komplizierter Datenstrukturen und C bietet sowohl strukturierten Code als auch direkten Speicherzugriff.) Diese Prozeduren sind in einer Quick-Bibliothek zum Gebrauch im Programm kompiliert und gebunden. Es ist ebenfalls möglich, ein separates Modul zur Prüfung der Quick-Bibliothekprozeduren zu schreiben, das der Prüfung anderer Prozeduren ähnlich ist.

7.9 Kompilieren und Binden von Modulen

Das Endprodukt Ihrer Programmierarbeit besteht normalerweise aus einer selbständigen *.exe*-Datei. Eine solche Datei läßt sich in QuickBASIC durch Laden sämtlicher Module eines Programmes und anschließendes Wählen des Befehls **EXE-Datei erstellen** aus dem Menü **Ausführen** anlegen.

Sie können Module auch mit Hilfe des BC-Befehlszeilen-Compilers kompilieren und dann den Objektcode anhand von LINK kombinieren. Objektdateien aus Codes, die in anderen Sprachen geschrieben sind, können gleichzeitig eingebunden werden.

Hinweis Beim Gebrauch des Befehls **EXE-Datei erstellen**, werden alle Modul-Ebenen-Codes und die jeweiligen geladenen Prozeduren in der *.exe*-Datei eingeschlossen, unabhängig davon, ob das Programm diesen Code verwendet oder nicht. Wenn Sie das Programm so kompakt wie möglich gestalten möchten, sollten Sie sämtliche unnötige Modul-Ebenen-Codes und Prozeduren vor der Kompilierung entladen. Dieselbe Regel, sämtliche unbenutzte Codes aus den Dateien zu entfernen, ist bei der Kompilierung anhand der BC-Befehlszeile gültig.

7.10 Quick-Bibliotheken

Obwohl Microsoft Quick-Bibliotheken keine Module darstellen, ist es wichtig, ihre Beziehung zu den Modulen zu verstehen.

Eine Quick-Bibliothek enthält ausschließlich Prozeduren. Diese Prozeduren können sowohl in QuickBASIC, als auch in anderen Microsoft-Sprachen (C, Pascal, FORTRAN und MASM) geschrieben sein.

Eine Quick-Bibliothek enthält nur kompilierte Codes. (Module enthalten QuickBASIC-Quellencodes.) Eine Quick-Bibliothek entsteht durch Einbinden kompilierten Objektcodes (*.obj*-Dateien). Der Code einer Quick-Bibliothek kann aus der Kombination jeglicher Microsoft-Sprachen bestehen. Anhang H, "Erstellung und Verwendung der Quick-Bibliotheken" beschreibt die Erstellung von Quick-Bibliotheken aus Objektcodes und die Ergänzung bestehender Quick-Bibliotheken mit neuen Objektcodes.

Quick-Bibliotheken erfüllen mehrere Zwecke:

- Sie stellen eine Schnittstelle zwischen QuickBASIC und anderen Sprachen zur Verfügung.
- Sie erlauben Entwicklern, Eigentumssoftware zu schützen. Überarbeitungen oder Hilfsprogramme können als Quick-Bibliotheken verteilt werden, ohne den Quellencode aufzudecken.
- Sie werden schneller geladen und sind gewöhnlich kleiner als die Module. Läßt sich ein umfangreiches Programm mit zahlreichen Modulen nur langsam laden, verbessert die Umwandlung der Nichthauptmodule in eine Quick-Bibliothek die Ladezeit.

Es ist jedoch zu beachten, daß die Arbeit mit den Modulen während der Entwicklung am einfachsten ist, da Module nach jeder Bearbeitung sofort lauffähig sind und die Quick-Bibliothek nicht neu erstellt werden muß. Wenn Sie Ihre QuickBasic Prozeduren in einer Quick-Bibliothek haben wollen, warten Sie mit dem Erstellen der Bibliothek, bis diese vollständig und völlig fehlerfrei sind.

Jeglicher Modul-Ebenen-Code der Datei, aus der eine Quick-Bibliothek erstellt wurde, ist automatisch darin eingebunden. Da jedoch andere Module keinen Zugriff zu diesem Code haben, ist er überflüssig und nimmt nur Platz ein. Vergewissern Sie sich vor der Umwandlung eines Moduls in eine Quick-Bibliothek, daß alle Modul-Ebenen-Anweisungen (mit Ausnahme der von den Prozeduren verwendeten Fehler- oder Ereignisbehandlungsroutinen und Vereinbarungen) entfernt wurden.

Hinweis Quick-Bibliotheken sind nicht in *.mak*-Dateien eingeschlossen und müssen mit einer /L-Option geladen werden, wenn Sie QuickBASIC laufen lassen. Eine Quick-Bibliothek hat die Erweiterung *.qlb*. Während des Erstellungsvorgangs der Quick-Bibliothek wird eine andere Bibliothekdatei mit der Erweiterung *.lib* angelegt. Diese Datei enthält den gleichen Code wie die Quick-Bibliothek in einer Form, die es ihr erlaubt in den Rest des Programmes eingebunden zu werden, und somit eine selbständig ausführbare Anwendung zu erstellen.

Werden Quick-Bibliotheken beim Verteilen von Eigentumscode (beispielsweise Prozeduren für die Datenmanipulation) verwendet, achten Sie darauf, daß die *.lib*-Dateien mitgeliefert werden, damit die Kunden in der Lage sind, selbständige Anwendungen unter Verwendung dieser Prozeduren zu erstellen. Andernfalls sind sie ausschließlich auf Anwendungen innerhalb der QuickBASIC-Umgebung beschränkt.

7.10.1 Erstellen von Quick-Bibliotheken

Eine Quick-Bibliothek von QuickBASIC-Prozeduren kann anhand des Befehls **Bibliothek erstellen** aus dem Menü **Ausführen** angelegt werden. Die erstellte Quick-Bibliothek enthält sämtliche im Moment geladene Prozeduren, unabhängig davon, ob diese vom Programm aufgerufen werden oder nicht. (Sie enthält auch sämtlichen Modul-Ebenen-Code.) Wenn Sie die Quick-Bibliothek kompakt gestalten möchten, sollten Sie zuerst alle unbenutzten Prozeduren und unnötigen Modul-Ebenen-Code entfernen.

Sie können eine beliebige Anzahl von Quick-Bibliotheken anlegen, die die gewünschte Kombination von Prozeduren enthalten. Es ist jedoch nicht möglich, mehrere Quick-Bibliotheken gleichzeitig in QuickBASIC zu laden. (Im allgemeinen werden anwendungsspezifische Quick-Bibliotheken angelegt, die nur die von einem bestimmten Programm benötigten Prozeduren enthalten.) Umfangreiche Quick-Bibliotheken lassen sich anlegen, indem Sie zahlreiche Module laden und danach den Befehl **Bibliothek erstellen** benutzen.

Eine andere Möglichkeit zum Erstellen einer Quick-Bibliothek besteht in der Kompilierung eines oder mehrerer Module mit dem Befehl BC und des anschließenden Bindens zu Objektcodedateien. Quick-Bibliotheken mit Prozeduren, die in anderen Sprachen geschrieben sind, werden auf dieselbe Weise angelegt. Das Binden ist nicht auf eine einzige Sprache begrenzt; die Objektcodedateien einer beliebigen Anzahl von Microsoft-Sprachen können in einer Quick-Bibliothek kombiniert werden. Anhang H, "Erstellung und Verwendung der Quick-Bibliotheken", beschreibt die Umwandlung von Objektcodedateien (*.obj*) in Quick-Bibliotheken.

7.11 Ratschläge für ein fachgerechtes Programmieren mit Modulen

Module können in jeder Art und Weise verwendet werden, die der Verbesserung des Programmes dient oder bei der Organisation der Arbeit behilflich ist. Folgende Hinweise werden nur als Ratschläge angeboten:

1. Überdenken und organisieren Sie Ihr Projekt vor Beginn der Arbeit.

   Am Anfang eines Projektes listen Sie zuerst alle von den Prozeduren auszuführenden Operationen auf. Gehen Sie dann die eigene Prozedurenbibliothek durch, um nach den Prozeduren zu suchen, die in ihren gegenwärtigen Zustand oder mit geringfügigen Änderungen anwendbar sind. Verlieren Sie keine Zeit mit dem "Neuerfinden des Schießpulvers".

2. Schreiben Sie allgemeine Prozeduren mit weitgehenden Anwendungen.

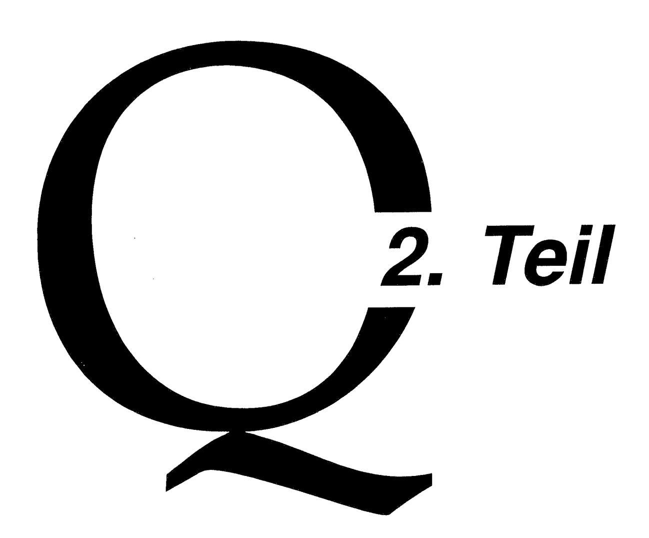
   Versuchen Sie Prozeduren zu schreiben, die in einer Reihe von unterschiedlichen Programmen zur Anwendung kommen, ohne diese dabei unnötig kompliziert zu gestalten. Eine gute Prozedur ist ein einfaches, gut eingestelltes Werkzeug, kein scharfes Schweizer Messer.

   Es kann manchmal angebracht sein, eine bestehende Prozedur für den Einsatz in einem neuen Programm umzuändern. Dieses Verfahren kann seinerseits das Ändern von bereits geschriebenen Programmen erfordern, aber die Mühe lohnt sich, wenn die überarbeitete Prozedur leistungsfähiger ist und einen weiteren Anwendungsbereich hat.

3. Behalten Sie beim Erstellen der eigenen Prozedurmodule die logisch getrennten Prozeduren in separaten Modulen.

   Es ist sinnvoll, Prozeduren der Zeichenkettenmanipulation einem Modul, Prozeduren der Matrizenbehandlung einem anderen und Prozeduren der Datenübertragung einem dritten Modul einzugliedern. Dieser Aufbau vermeidet Mißverständnisse und erleichtert das Auffinden der benötigten Prozedur.

2. Teil: BASIC-Grundbegriffe



2. Teil

IF...THEN...ELSE-Anweisungen

Ermöglichen die bedingte Ausführung, die auf der Auswertung eines Booleschen Ausdrucks beruht.

Syntax 1 (EINZEILIG) **IF** *Boolescher Ausdruck* **THEN** *Dannteil* [ELSE *sonst-Teil*]

Syntax 2 (BLOCK) **IF** Boolescher Ausdruck1 **THEN**

```
IF Boolescher Ausdruck1 THEN
    [Anweisungsblock-1]
[ELSEIF Boolescher Ausdruck2 THEN
    [Anweisungsblock-2]]
.
.
.
[ELSE
    [Anweisungsblock-n]]
END IF
```

BASIC-Grundbegriffe

Der 2. Teil bietet eine zusammengefaßte Referenzanleitung zu sämtlichen in dieser BASIC-Version verwendeten Anweisungen und Funktionen.

Kapitel 8 faßt in alphabetischer Reihenfolge der Schlüsselwörter die Aktionen aller Anweisungen und Funktionen zusammen, einschließlich der Syntaxzeilen, die die korrekte Form der Anweisung kennzeichnen. Dieser Abschnitt soll beim Programmieren als Gedächtnisstütze für die Arbeitsweise der Anweisungen und Funktionen dienen.

Kapitel 9 besteht aus Referenztabellen, die häufig verwendete nach Programmierthemen angeordnete Anweisungen und Funktionen aufführen. Diese Tabellen stimmen mit der Gliederung der ersten 6 Kapitel dieses Handbuchs überein und sollen dem Benutzer helfen, alternative Möglichkeiten zur Lösung einer bestimmten Programmieraufgabe zu entdecken.

Kapitelverzeichnis

8 Zusammenfassung der Anweisungen und Funktionen
9 Quick-Referenztabellen

8 Zusammenfassung der Anweisungen und Funktionen

Dieses Kapitel faßt QuickBASIC-Anweisungen und -Funktionen zusammen. Jedem Anweisungs- oder Funktionsnamen folgt eine Beschreibung des ausgeführten Vorganges und eine Syntaxzeile. Die Syntaxzeile führt genau die zur Verwendung der Anweisung erforderliche Eingabe auf.

Die in den Syntaxzeilen auftretenden typographischen Vereinbarungen werden in der Einleitung dieses Handbuches eingehend beschrieben. Im allgemeinen sind die genau zu übernehmenden Elemente in Fettdruck und die Platzhalter für einzugebende Informationen in Kursivschrift aufgeführt. Optionen stehen in eckigen Klammern.

Der QB-Ratgeber (die On-Line-Sprachhilfe von QuickBASIC) erteilt folgende Auskünfte über jede Anweisung oder Funktion:

- Eine Zusammenfassung der Tätigkeit und Syntax jeder Anweisung (QuickSCREEN).
- Einzelheiten über die Verwendung jeder Anweisung, einschließlich der Erklärung der Platzhalter.
- Ein oder mehrere Programmbeispiele, die die Anwendung der Anweisung veranschaulichen.

Sie erhalten Zugriff zu diesen Informationen, indem Sie den Cursor auf einem beliebigen, vom QuickBASIC-Arbeitsfenster angezeigten BASIC-Schlüsselwort positionieren und dann F1 drücken.

Die in diesem Kapitel dargestellten Anweisungen und Funktionen werden themenweise in Kapitel 9, "Quick-Referenztabellen", gruppiert. Die für eine bestimmte Programmieraufgabe geeigneten Anweisungen sind in Kapitel 9 zu finden.

ABS-Funktion

Gibt den absoluten Wert eines numerischen Ausdruckes zurück.

Syntax **ABS** (*Numerischer Ausdruck*)

ASC-Funktion

Gibt einen numerischen Wert zurück, der den ASCII-Code für das erste Zeichen in einem Zeichenkettenausdruck darstellt.

Syntax **ASC** (*Zeichenkettenausdruck*)

ATN-Funktion

Gibt den Arkustangens eines numerischen Ausdruckes zurück (den Winkel, dessen Tangens dem numerischen Ausdruck entspricht).

Syntax **ATN** (*Numerischer Ausdruck*)

BEEP-Anweisung

Sendet ein Tonsignal an den Lautsprecher.

Syntax **BEEP**

BLOAD-Anweisung

Lädt eine von **BSAVE** erstellte Speicherbilddatei von einer Eingabedatei oder einem Eingabegerät in den Speicher.

Syntax **BLOAD** *Dateiangabe* [, *Offset*]

BSAVE-Anweisung

Übergibt den Inhalt eines Speicherbereiches an eine Ausgabedatei oder ein Ausgabegerät.

Syntax **BSAVE** *Dateiang, Offset, Länge*

CALL-Anweisung (BASIC-Prozeduren)

Übergibt die Kontrolle an eine BASIC SUB.

Syntax 1 **CALL** *Name* [(*Argumentenliste*)]

Syntax 2 *Name* [*Argumentenliste*]

CALL-, CALLS-Anweisung (Nicht-BASIC-Prozeduren)

Übergibt die Kontrolle an eine in einer anderen Sprache geschriebene Prozedur.

Syntax 1 **CALL** *Name* [(*Aufruf-Argumentenliste*)]

Syntax 2 *Name* [*Aufruf-Argumentenliste*]

Syntax 3 **CALLS** *Name* [(*Aufrufe-Argumentenliste*)]

CALLINT86OLD-Anweisungen

Ermöglicht dem Programm, DOS-Systemaufrufe auszuführen.

Syntax **CALLINT86OLD** (*Int_nr, In_Datenfeld( ), Aus_Datenfeld( )*)
CALLINT86XOLD (*Int_nr, In_Datenfeld( ), Aus_Datenfeld( )*)

CALL ABSOLUTE-Anweisung

Übergibt die Kontrolle an eine Prozedur in Maschinensprache.

Syntax **CALL ABSOLUTE** ([*Argumentenliste*,] *Ganzzahlvariable*)

CALL INTERRUPT-Anweisungen

Ermöglichen BASIC-Programmen, DOS-Systemaufrufe auszuführen.

Syntax **CALL INTERRUPT** (*Interruptnum, Inreg, Ausreg*)
CALL INTERRUPTX (*Interruptnum, Inreg, Ausreg*)

CDBL-Funktion

Wandelt einen numerischen Ausdruck in eine Zahl doppelter Genauigkeit um.

Syntax **CDBL** (*Numerischer Ausdruck*)

CHAIN-Anweisung

Übergibt die Kontrolle aus dem laufenden Programm an ein anderes Programm.

Syntax **CHAIN** *Dateiangabe*

CHDIR-Anweisung

Ändert das aktuelle Standardverzeichnis für das angegebene Laufwerk.

Syntax **CHDIR** *Pfadangabe*

CHR$-Funktion

Gibt eine aus einem Zeichen bestehende Zeichenkette zurück, dessen ASCII-Code das Argument darstellt.

Syntax **CHR$** (*Code*)

CINT-Funktion

Wandelt einen numerischen Ausdruck durch Runden der Stellen hinter dem Komma in eine Ganzzahl um.

Syntax **CINT** (*Numerischer Ausdruck*)

CIRCLE-Anweisung

Zeichnet eine Ellipse oder einen Kreis mit dem angegebenen Mittelpunkt und Radius.

Syntax **CIRCLE** [**STEP**] (*x,y*), *Radius*[,[*Farbe*][,[*Anfang*][, [*Ende*][, *Aspekt*]]]]

CLEAR-Anweisung

Reinitialisiert alle Programmvariablen, schließt Dateien und legt die Stapelgröße (Stack) fest.

Syntax **CLEAR** [,, *Stapel*]

CLNG-Funktion

Wandelt einen numerischen Ausdruck durch Runden der Stellen hinter dem Komma in eine lange (4-Byte-) Ganzzahl um.

Syntax **CLNG** (*Numerischer Ausdruck*)

CLOSE-Anweisung

Beendet E/A in/aus eine(r) Datei oder ein(em) Gerät.

Syntax **CLOSE** [[#] *Dateinummer* [,[#] *Dateinummer*]...]

CLS-Anweisung

Löscht den Bildschirm.

Syntax **CLS** [{0 | 1 | 2}]

COLOR-Anweisung

Wählt die Anzeigefarben aus.

Syntax	**COLOR** [*Vordergrund*][, [*Hintergrund*],_ [*Rahmen*]]	Bildschirmmodus 0
	COLOR [*Hintergrund*][, *Palette*]	Bildschirmmodus 1
	COLOR [*Vordergrund*][, *Hintergrund*]	Bildschirmmodi 7-10
	COLOR [*Vordergrund*]	Bildschirmmodi 12-13

COM-Anweisungen

Aktivieren, deaktivieren oder sperren die Ereignisverfolgung der Datenübertragungstätigkeit an einem gegebenen Anschluß.

Syntax **COM**(*n*) **ON**
COM(*n*) **OFF**
COM(*n*) **STOP**

COMMAND$-Funktion

Gibt die zum Aufruf des Programms benutzte Befehlszeile zurück.

Syntax **COMMAND$**

COMMON-Anweisung

Definiert globale Variablen, die zwischen Modulen geteilt oder mit anderen Programmen verkettet werden.

Syntax **COMMON** [**SHARED**][/*Blockname*/] *Variablenliste*

CONST-Anweisung

Deklariert die an Stelle von numerischen oder Zeichenkettenwerten zu verwendenden Konstanten.

Syntax **CONST** *Konstantenname* = *Ausdruck* [, *Konstantenname* = *Ausdruck*]...

COS-Funktion

Gibt den Kosinus eines im Bogenmaß angegebenen Winkels zurück.

Syntax **COS** (*Numerischer Ausdruck*)

CSNG-Funktion

Wandelt einen numerischen Ausdruck in einen Wert einfacher Genauigkeit um.

Syntax **CSNG** (*Numerischer Ausdruck*)

CSRLIN-Funktion

Gibt die gegenwärtige Position des Cursors in der Zeile an.

Syntax **CSRLIN**

CVI-,CVS-,CVL-,CVD-Funktionen

Wandeln Zeichenketten, die numerische Werte enthalten, in Zahlen um.

Syntax **CVI** (*2-Byte-Zeichenkette*)
CVS (*4-Byte-Zeichenkette*)
CVL (*4-Byte-Zeichenkette*)
CVD (*8-Byte-Zeichenkette*)

CVSMBF-,CVDMBF-Funktionen

Wandeln Zeichenketten, die Zahlen im Microsoft-Binär-Format enthalten, in Zahlen mit IEEE-Format um.

Syntax **CVSMBF** (*4-Byte-Zeichenkette*)
CVDMBF (*8-Byte-Zeichenkette*)

DATA-Anweisung

Speichert die von den **READ**-Anweisungen eines Programmes verwendeten numerischen und Zeichenkettenkonstanten.

Syntax **DATA** *Konstante1* [,*Konstante2*]...

DATE$-Funktion

Gibt eine Zeichenkette mit dem aktuellen Datum zurück.

Syntax **DATE$**

DATE$-Anweisung

Setzt das aktuelle Datum.

Syntax **DATE$** = *Zeichenkettenausdruck*

DECLARE-Anweisung (BASIC-Prozeduren)

Deklariert den Bezug zu BASIC-Prozeduren und ruft eine Argumenttyp-Prüfung auf.

Syntax **DECLARE** {**FUNCTION** | **SUB**} *Name*
[([*Parameterliste*])]

DECLARE-Anweisung (Nicht-BASIC-Prozeduren)

Deklariert Aufrufsequenzen für externe, in andere Sprachen geschriebene Prozeduren.

Syntax 1 **DECLARE FUNCTION** *Name*[**CDECL**]
[**ALIAS**"*Aliasname*"][([*Parameter-liste*])]

Syntax 2 **DECLARE SUB** *Name* [**CDECL**]
[**ALIAS**"*Aliasname*"][([*Parameterliste*])]

DEF FN-Anweisung

Definiert und bezeichnet eine Funktion.

Syntax 1 **DEF FN** *Name* [(*Parameterliste*)] = *Ausdruck*

Syntax 2 **DEF FN** *Name* [(*Parameterliste*)]

.
.
.
FN *Name* = *Ausdruck*
.
.
.
END DEF

DEF SEG-Anweisung

Setzt die aktuelle Segmentadresse für eine nachfolgende **PEEK**-Funktion oder eine **BLOAD**-, **BSAVE**-, **CALL ABSOLUTE** oder **POKE**-Anweisung.

Syntax **DEF SEG** [=*Adresse*]

DEFTyp-Anweisungen

Setzen den Standarddatentyp für Variablen, **DEF FN**-Funktionen und **FUNCTION**-Prozeduren.

Syntax
DEFINT *Buchstabenbereich* [, *Buchstabenbereich*]...
DEFSNG *Buchstabenbereich* [, *Buchstabenbereich*]...
DEFDBL *Buchstabenbereich* [, *Buchstabenbereich*]...
DEFLNG *Buchstabenbereich* [, *Buchstabenbereich*]...
DEFSTR *Buchstabenbereich* [, *Buchstabenbereich*]...

DIM-Anweisung

Deklariert eine Variable und weist Speicherplatz zu.

Syntax **DIM** [**SHARED**] *Variable*[(*Indizes*)][**AS** *Typ*][,*Variable* [(*Indizes*)] [**AS** *Typ*]]...

DO...LOOP-Anweisungen

Wiederholen Anweisungsblöcke solange eine Bedingung wahr ist oder bis eine Bedingung wahr wird.

Syntax 1 **DO**
[Anweisungsblock]
LOOP [{**WHILE** | **UNTIL**} *Boolescher Ausdruck*]

Syntax 2 **DO** [{**WHILE** | **UNTIL**} *Boolescher Ausdruck*]
[Anweisungsblock]
LOOP

DRAW-Anweisung

Zeichnet ein durch einen *Zeichenkettenausdruck* definiertes Objekt.

Syntax **DRAW** *Zeichenkettenausdruck*

END-Anweisung

Beendet ein BASIC-Programm, eine BASIC-Prozedur oder einen BASIC-Block.

Syntax **END**[{**DEF** | **FUNCTION** | **IF** | **SELECT**|**SUB** | **TYPE**}]

ENVIRON$-Funktion

Stellt eine Umgebungszeichenkette aus der DOS-Umgebungszeichenketten-Tabelle bereit.

Syntax **ENVIRON$** (*Umgebungszeichenkette*)
ENVIRON$ (*n*)

ENVIRON-Anweisung

Ändert einen Parameter in der DOS-Tabelle der Umgebungszeichenketten.

Syntax **ENVIRON** *Zeichenkettenausdruck*

EOF-Funktion

Prüft auf die Bedingung Dateiende.

Syntax **EOF** (*Dateinummer*)

ERASE-Anweisung

Reinitialisiert die Elemente eines statischen Datenfeldes; hebt die Speicherzuordnung dynamischer Felder auf.

Syntax **ERASE** *Datenfeldname* [, *Datenfeldname...*]

ERDEV-,ERDEV$-Funktionen

Erteilt gerätespezifische Statusinformationen nach einem Fehler.

Syntax **ERDEV**
ERDEV$

ERR-,ERL-Funktionen

Geben den Fehlerstatus an.

Syntax **ERR**
ERL

ERROR-Anweisung

Simuliert das Auftreten eines BASIC-Fehlers und erlaubt dem Benutzer die Definition von Fehlercodes.

Syntax **ERROR** *Ganzzahlausdruck*

EXIT-Anweisung

Beendet eine **DEF FN**-Funktion, eine **DO...LOOP**- oder **FOR...NEXT**-Schleife, **FUNCTION** oder **SUB**.

Syntax **EXIT** {**DEF** | **DO** | **FOR** | **FUNCTION** | **SUB**}

EXP-Funktion

Berechnet die Exponentialfunktion.

Syntax **EXP**(*x*)

FIELD-Anweisung

Weist Variablen in einem Direktzugriffs-Dateipuffer Speicherplatz zu.

Syntax **FIELD** [#] *Dateinummer*, *Datenfeldlänge* **AS** *Zeichenkettenvariable...*

FILEATTR-Funktion

Erteilt Informationen über eine geöffnete Datei.

Syntax **FILEATTR** (*Dateinummer*, *Attribut*)

FILES-Anweisung

Gibt die Namen der auf einer angegebenen Diskette/Festplatte vorhandenen Dateien aus.

Syntax **FILES** [*Dateiangabe*]

FIX-Funktion

Gibt den abgeschnittenen ganzzahligen Teil von *x* zurück.

Syntax **FIX**(*x*)

FOR...NEXT-Anweisung

Führt eine Folge von Anweisungen in einer Schleife aus. Die Anzahl der Schleifendurchläufe ist angegeben.

Syntax **FOR** *Zähler* = *Start* **TO** *Ende* [**STEP** *Schrittweite*]

.
.
.

NEXT [*Zähler* [, *Zähler*...]]

FRE-Funktion

Gibt die Größe des zur Verfügung stehenden Speichers an.

Syntax 1 **FRE** (*Numerischer Ausdruck*)

Syntax 2 **FRE** (*Zeichenkettenausdruck*)

FREEFILE-Funktion

Gibt die nächste freie BASIC-Dateinummer an.

Syntax **FREEFILE**

FUNCTION-Anweisung

Deklariert Namen, Parameter und Code, die den Rumpf einer **FUNCTION**-Prozedur bilden.

Syntax **FUNCTION** *Name* [(*Parameter-Liste*)][**STATIC**]

.
.
.
Name = *Ausdruck*
.
.
.
END FUNCTION

GET-Anweisung - Datei-E/A

Liest von einer Diskettendatei in einen Puffer mit Direktzugriff oder in eine Variable ein.

Syntax **GET** [#] *Dateinummer* [,[*Datensatznummer*][,*Variable*]]

GET-Anweisung-Graphik

Speichert graphische Bilder vom Bildschirm.

Syntax **GET** [**STEP**] (*x1,y1*) - [**STEP**] (*x2,y2*), *Datenfeldname* [(*Indizes*)]

GOSUB...RETURN-Anweisungen

Verzweigt zu einer Unterroutine und kehrt von dort zurück.

Syntax **GOSUB** (*Zeilenmarke* | Zeilennummer)

.
.
.
RETURN [*Zeilenmarke2* | *Zeilennummer2*]

GOTO-Anweisung

Verzweigt bedingungslos zu der angegebenen Zeile.

Syntax **GOTO** {*Zeilenmarke* | *Zeilennummer*}

HEX$-Funktion

Gibt eine Zeichenkette an, die den Hexadezimalwert des dezimalen Arguments *Ausdruck* darstellt.

Syntax **HEX$** (*Ausdruck*)

IF...THEN...ELSE-Anweisungen

Ermöglichen die bedingte Ausführung, die auf der Auswertung eines Booleschen Ausdrucks beruht.

Syntax 1 **(EINZEILIG)** **IF** *Boolescher Ausdruck* **THEN** *Dannteil* [**ELSE** *Sonst-Teil*]

Syntax 2 **(BLOCK)** **IF** *Boolescher Ausdruck1* **THEN**
  [*Anweisungsblock-1*]
[**ELSEIF** *Boolescher Ausdruck2* **THEN**
  [*Anweisungsblock-2*]]
.
.
.
[**ELSE**
  [*Anweisungsblock-n*]]
END IF

INKEY$-Funktion

Liest ein Zeichen von der Tastatur.

Syntax **INKEY$**

INP-Funktion

Gibt das von einem E/A-Anschluß eingelesene Byte an.

Syntax **INP** (*Anschluß*)

INPUT$-Funktion

Gibt eine Folge von Zeichen an, die aus der angegebenen Datei gelesen werden.

Syntax **INPUT$** (*n*[,[#] *Dateinummer*])

INPUT-Anweisung

Ermöglicht die Tastatureingabe während der Programmausführung.

Syntax **INPUT** [;]["*Anfragezeichenkette*"{; | ,}] *Variablenliste*

INPUT#-Anweisung

Liest Daten aus einem sequentiellen Gerät oder einer sequentiellen Datei ein und weist sie Variablen zu.

Syntax **INPUT#** *Dateinummer*, *Variablenliste*

INSTR-Funktion

Übergibt die Zeichenposition des ersten Vorkommens einer Zeichenkette in einer anderen Zeichenkette.

Syntax **INSTR** ([Start,] *Zeichenkettenausdruck1*, *Zeichenkettenausdruck2*)

INT-Funktion

Gibt die größte Ganzzahl kleiner oder gleich *Numerischer Ausdruck* an.

Syntax **INT** (*Numerischer Ausdruck*)

IOCTL$-Funktion

Empfängt eine Kontrolldaten-Zeichenkette von einem Gerätetreiber.

Syntax **IOCTL$** ([#] *Dateinummer*)

IOCTL-Anweisung

Überträgt eine Steuerdaten-Zeichenkette an einen Gerätetreiber.

Syntax **IOCTL** [#] *Dateinummer*, *Zeichenkette*

KEY-Anweisungen

Weisen den Funktionstasten Zeichenkettenwerte als Tastenbelegung zu, zeigen dann die Werte an und schalten die Anzeigezeile für die FUNCTION-Taste ein oder aus.

Syntax **KEY***n*, *Zeichenkettenausdruck*
KEY LIST
KEY ON
KEY OFF

KEY(n)-Anweisungen

Starten und beenden die Verfolgung angegebener Tasten.

Syntax **KEY(***n***) ON**
KEY(*n***) OFF**
KEY(*n***) STOP**

KILL-Anweisung

Löscht eine Datei von einer Diskette.

Syntax **KILL** *Dateiangabe*

LBOUND-Funktion

Liefert die untere Grenze (den kleinsten vorhandenen Index) für die angegebene Dimension eines Datenfeldes.

Syntax **LBOUND** (*Datenfeld* [,*Dimension*])

LCASE$-Funktion

Gibt einen Zeichenkettenausdruck mit sämtlichen Zeichen als Kleinbuchstaben an.

Syntax **LCASE$** (*Zeichenkettenausdruck*)

LEFT$-Funktion

Gibt eine Zeichenkette an, die aus den *n* äußerst links stehenden Zeichen einer Zeichenkette besteht.

Syntax **LEFT$** (*Zeichenkettenausdruck*, *n*)

LEN-Funktion

Gibt die Anzahl der Zeichen in einer Zeichenkette oder die Anzahl der von einer Variablen benötigten Bytes an.

Syntax **LEN** (*Zeichenkettenausdruck*)
LEN (*Variable*)

LET-Anweisung

Weist einer Variablen den Wert eines Ausdruckes zu.

Syntax [**LET**] *Variable* = *Ausdruck*

LINE-Anweisung

Zeichnet eine Gerade oder ein Rechteck auf dem Bildschirm.

Syntax **LINE** [[**STEP**] (*x1*,*y1*)] - [**STEP**] (*x2*,*y2*) [,[*Farbe*][,[**B**[**F**]][, *Struktur*]]]

LINE INPUT-Anweisung

Liest eine ganze Zeile (bis zu 255 Zeichen) in eine Zeichenkettenvariable ein, ohne dabei Begrenzer zu verwenden.

Syntax **LINE INPUT** [;]["*Anfragezeichenkette*";] *Zeichenkettenvariable*

LINE INPUT#-Anweisung

Liest eine ganze Zeile ohne Begrenzer aus einer sequentiellen Datei in eine Zeichenkettenvariable ein.

Syntax **LINE INPUT** *#Dateinummer*, *Zeichenkettenvariable*

LOC-Funktion

Gibt die aktuelle Position in der Datei an.

Syntax **LOC** (*Dateinummer*)

LOCATE-Anweisung

Bewegt den Cursor zur angegebenen Position.

Syntax **LOCATE** [*Zeile*][,[*Spalte*][,[*Cursor*][,[*Anfang*, *Ende*]]]]

LOCK...UNLOCK-Anweisung

Steuert den Zugriff anderer Prozesse auf eine gesamte oder einen Teil einer geöffneten Datei.

Syntax **LOCK** [#] *Dateinummer* [,{*Datensatz* | [*Anfang*] **TO** *Ende*}]

.
.
.

UNLOCK [#] *Dateinummer* [,{*Datensatz* | [*Anfang*] **TO** *Ende*}]

LOF-Funktion

Gibt die Länge der genannten Datei in Bytes an.

Syntax **LOF** (*Dateinummer*)

LOG-Funktion

Gibt den natürlichen Logarithmus eines numerischen Ausdrucks zurück.

Syntax **LOG**(*n*)

LPOS-Funktion

Gibt die aktuelle Druckkopfposition des Zeilendruckers im Druckerpuffer an.

Syntax **LPOS**(*n*)

LPRINT-, LPRINT USING-Anweisungen

Drucken Daten auf dem Drucker LPT1 aus:

Syntax 1 **LPRINT** [*Ausdrucksliste*][{; | ,}]

Syntax 2 **LPRINT USING** *Formatzeichenkette*; *Ausdrucksliste*[{; | ,}]

LSET-Anweisung

Bringt Daten aus dem Speicher in einen Direktzugriffs-Dateipuffer (als Vorbereitung auf eine **PUT**-Anweisung), kopiert eine Datensatzvariable in eine andere oder ordnet den Wert einer Zeichenkette in einer Zeichenkettenvariablen linksbündig an.

Syntax **LSET** {*Zeichenkettenvariable* = *Zeichenkettenausdruck* |_
Zeichenkettenausdruck1 = *Zeichenkettenausdruck2* }

LTRIM$-Funktion

Gibt die Kopie einer Zeichenkette an, deren führende Leerzeichen entfernt sind.

Syntax **LTRIM$** (*Zeichenkettenausdruck*)

MID$-Funktion

Gibt eine Teilzeichenkette einer Zeichenkette an.

Syntax **MID$** (*Zeickenkettenausdruck*, *Anfang* [, *Länge*])

MID$-Anweisung

Ersetzt einen Teil einer Zeichenkettenvariablen durch eine andere Zeichenkette.

Syntax **MID$** (*Zeichenkettenvariable*, *Anfang* [, *Länge*]) = *Zeichenkettenausdruck*

MKD$-,MKI$-MKL$-,MKS$-Funktionen

Wandeln numerische Werte in Zeichenkettenwerte um.

Syntax **MIK$** (*Ganzzahlausdruck*)
MKS$ (*Ausdruck einfacher Genauigkeit*)
MKL$ (*langganzzahliger Ausdruck*)
MKD$ (*Ausdruck doppelter Genauigkeit*)

MKDIR-Anweisung

Erstellt ein neues Verzeichnis.

Syntax **MKDIR** *Pfadname*

MKSMBF$-,MKDMBF$-Funktionen

Wandeln eine Zahl im IEEE-Format in eine Zeichenkette um, die eine Microsoft-Binär-Formatzahl enthält.

Syntax **MKSMBF$** (*Ausdruck einfacher Genauigkeit*)
MKDMBF$ (*Ausdruck doppelter Genauigkeit*)

NAME-Anweisung

Ändert den Namen einer Diskettendatei oder eines Verzeichnisses.

Syntax **NAME** *alter Dateiname* **AS** *neuer Dateiname*

OCT$-Funktion

Liefert eine Zeichenkette, die den Oktalwert des numerischen Arguments darstellt.

Syntax **OCT$** (*Numerischer Ausdruck*)

ON ERROR-Anweisung

Aktiviert die Fehlerbehandlung und gibt die erste Zeile der Fehlerbehandlungsroutine an.

Syntax **ON ERROR GOTO** *Zeile*

ON Ereignis-Anweisung

Markiert die erste Zeile einer Ereignisverfolgungs-Unterroutine.

Syntax **ON** *Ereignis* **GOSUB** {*Zeilennummer* | *Zeilenmarke*}

ON UEVENT GOSUB-Anweisung

Definiert die Fehlerbehandlung für ein benutzerdefiniertes Ereignis.

Syntax **ON UEVENT GOSUB** {*Zeilennummer* | *Zeilenmarke*}

ON...GOSUB-,ON...GOTO-Anweisungen

Verzweigen, je nach dem Wert eines Ausdruckes, zu einer von mehreren angegebenen Zeilen.

Syntax 1 **ON** *Ausdruck* **GOSUB** {*Zeilennummernliste* | *Zeilenmarkenliste*}

Syntax 2 **ON** *Ausdruck* **GOTO** {*Zeilennummernliste* | *Zeilenmarkenliste*}

OPEN-Anweisung

Ermöglicht E/A in eine Datei oder ein Gerät.

Syntax 1 **OPEN** *Datei* [**FOR** *Modus1*][**ACCESS** *Zugriff*][*Sperrtyp*] **AS** [#] *Dateinummer* [**LEN** = *Datensatzlänge*]

Syntax 2 **OPEN** *Modus2*, [#] *Dateinummer*,*Datei* [, *Datensatzlänge*]

OPEN COM-Anweisung

Öffnet und initialisiert einen Datenübertragungskanal für E/A.

Syntax **OPEN** "**COM***n*: *Optionsliste1 Optionsliste2*" [**FOR** *Modus*] **AS** [#] *Dateinummer* [**LEN** = *Datensatzlänge*]

OPTION BASE-Anweisung

Deklariert die untere Grenze für Datenfeldindizes.

Syntax **OPTION BASE***n*

OUT-Anweisung

Sendet ein Byte zu einem Maschinen-E/A-Anschluß.

Syntax **OUT** *Anschluß*, *Daten*

PAINT-Anweisung

Füllt eine Graphikfläche mit der angegebenen Farbe oder dem angegebenen Muster aus.

Syntax **PAINT** [**STEP**] (*x,y*) [,[*Malen*][,[*Randfarbe*][, *Hintergrund*]]]

PALETTE-,PALETTE USING-Anweisungen

Ändert eine oder mehrere Farben in der Palette.

Syntax **PALETTE** [*Attribute*, *Farbe*]
PALETTE USING *Datenfeldname* [(*Datenfeldindex*)]

PCOPY-Anweisung

Kopiert den Inhalt einer Bildschirmseite in eine andere.

Syntax **PCOPY** *Quellseite*, *Zielseite*

PEEK-Funktion

Liefert das in der angegebenen Speicheradresse gespeicherte Byte.

Syntax **PEEK** (*Adresse*)

PEN-Funktion

Liest die Lichtstift(Lightpen)-Koordinaten ein.

Syntax **PEN**(*n*)

PEN ON-,OFF- und STOP-Anweisungen

Aktiviert, deaktiviert oder unterbricht die Ereignisverfolgung für den Lichtstift.

Syntax **PEN ON**
PEN OFF
PEN STOP

PLAY-Funktion

Liefert die aktuelle Anzahl von Noten, die im Puffer für die Hintergrundmusik stehen.

Syntax **PLAY**(n)

PLAY-Anweisung

Spielt die durch eine Zeichenkette angegebene Musik.

Syntax **PLAY** *Befehlszeichenkette*

PLAY ON-,OFF- und STOP-Anweisungen

PLAY ON aktiviert die Ereignisverfolgung für die Musik, **PLAY OFF** deaktiviert und **PLAY STOP** unterbricht sie.

Syntax **PLAY ON**
PLAY OFF
PLAY STOP

PMAP-Funktion

Bildet logische Koordinaten-Ausdrücke auf physikalische Positionen oder physikalische Ausdrücke auf eine Position logischer Koordinaten ab.

Syntax **PMAP** (*Ausdruck*, *Funktion*)

POINT-Funktion

Liest die Farbnummer eines Bildpunktes vom Bildschirm oder gibt die Koordinaten des Bildpunktes zurück.

Syntax **POINT** (*x,y*)
POINT (*Nummer*)

POKE-Anweisung

Schreibt ein Byte in eine Speicherstelle.

Syntax **POKE** *Adresse*, *Byte*

POS-Funktion

Gibt die aktuelle horizontale Position des Cursors an.

Syntax **POS(0)**

PRESET-Anweisung

Stellt einen angegebenen Punkt auf dem Bildschirm dar.

Syntax **PRESET** [**STEP**] (*x-Koordinate*, *y-Koordinate*) [, *Farbe*]

PRINT-Anweisung

Gibt Daten auf dem Bildschirm aus.

Syntax **PRINT**[*Ausdrucksliste*][{, | ;}]

PRINT#-,PRINT#USING-Anweisungen

Schreiben Daten in eine sequentielle Datei.

Syntax **PRINT** # *Dateinummer*, [**USING** *Zeichenkettenausdruck*;] *Ausdrucksliste* [{, | ;}]

PRINT USING-Anweisung

Gibt Zeichen oder Zahlen in einem festgelegten Format aus.

Syntax **PRINT USING** *Formatzeichenkette*; *Ausdrucksliste* [{, | ;}]

PSET-Anweisung

Stellt einen Punkt auf dem Bildschirm dar.

Syntax **PSET** [**STEP**] (*x-Koordinate*, *y-Koordinate*) [, *Farbe*]

PUT-Anweisung - Datei-E/A

Schreibt aus einer Variablen oder einem Direktzugriffspuffer in eine Datei.

Syntax **PUT** [#] *Dateinummer*[,[*Datensatznummer*][,*Variable*]]
PUT [#] *Dateinummer* [,{*Datensatznummer* | *Datensatznummer*,*Variable* | , *Variable*}]

PUT-Anweisung - Graphik

Plaziert eine durch eine **GET**-Anweisung gespeicherte Graphik auf dem Bildschirm.

Syntax **PUT** [**STEP**] (*x*,*y*), *Datenfeldname* [(*Indizes*)][, *Aktionswort*]

RANDOMIZE-Anweisung

Initialisiert den Zufallszahlengenerator.

Syntax **RANDOMIZE** [*Ausdruck*]

READ-Anweisung

Liest Werte aus der **DATA**-Anweisung und weist diese Variablen zu.

Syntax **READ** *Variablenliste*

REDIM-Anweisung

Ändert die Größe, die einem mit **$DYNAMIC** deklarierten Datenfeld zugewiesen wurde.

Syntax **REDIM** [**SHARED**] *Variable* (*Indizes*) [**AS** *Typ*] [, *Variable* (*Indizes*) [**AS** *Typ*]]...

REM-Anweisung

Ermöglicht das Einfügen erläutender Anmerkungen in einem Programm.

Syntax 1 **REM** *Anmerkung*

Syntax 2 ' *Anmerkung*

RESET-Anweisung

Schließt alle Diskettendateien.

Syntax **RESET**

RESTORE-Anweisung

Ermöglicht es, **DATA**-Anweisungen ab einer angegebenen Zeile erneut einzulesen.

Syntax **RESTORE** [{*Zeilennummer* | *Zeilenmarke*}]

RESUME-Anweisung

Setzt die Programmausführung fort, nachdem eine Fehlerbehandlungsroutine ausgeführt wurde.

Syntax **RESUME** [0]
RESUME NEXT
RESUME {*Zeilennummer* | *Zeilenmarke*}

RETURN-Anweisung

Gibt die Kontrolle aus einer Unterroutine zurück.

Syntax **RETURN** [{*Zeilennummer* | *Zeilenmarke*}]

RIGHT$-Funktion

Liefert die *n* äußerst rechts stehenden Zeichen einer Zeichenkette.

Syntax **RIGHT$** (*Zeichenkettenausdruck, n*)

RMDIR-Anweisung

Löscht ein vorhandenes Verzeichnis.

Syntax **RMDIR** *Pfadname*

RND-Funktion

Gibt eine Zufallszahl einfacher Genauigkeit zwischen 0 und 1 an.

Syntax **RND**[(*n*)]

RSET-Anweisung

Bringt Daten aus dem Speicher in einen Direktzugriffs-Puffer (als Vorbereitung auf eine **PUT**-Anweisung) oder ordnet den Wert einer Zeichenkette in einer Zeichenkettenvariablen rechtsbündig an.

Syntax **RSET** *Zeichenkettenvariable = Zeichenkettenausdruck*

RTRIM$-Funktion

Gibt eine Zeichenkette aus, deren nachfolgende (rechte) Leerzeichen entfernt sind.

Syntax **RTRIM$** (*Zeichenkettenausdruck*)

RUN-Anweisung

Startet das gegenwärtig im Speicher vorhandene Programm erneut oder führt ein bestimmtes Programm aus.

Syntax **RUN** [{*Zeilennummer* | *Befehlszeile*}]

SADD-Function

Gibt die Adresse des angegebenen Zeichenausdruckes an.

Syntax **SADD** (*Zeichenkettenvariable*)

SCREEN-Funktion

Liest den ASCII-Wert eines Zeichens oder seine Farbe von der angegebenen Bildschirmposition.

Syntax **SCREEN** (*Zeile*, *Spalte*[, *Farbkennzeichen*])

SCREEN-Anweisung

Setzt die Bildschirmattribute.

Syntax **SCREEN** [*Modus*][, [*Farbschalter*]][,[*A-Seite*]][,[*V-Seite*]]

SEEK-Funktion

Gibt die aktuelle Dateiposition an.

Syntax **SEEK** (*Dateinummer*)

SEEK-Anweisung

Setzt die Position in einer Datei für den nächsten Lese- oder Schreibvorgang.

Syntax **SEEK** [#] *Dateinummer*, *Position*

SELECT CASE-Anweisung

Führt einen oder mehrere Anweisungsblöcke je nach dem Wert des Ausdruckes aus.

Syntax

```
SELECT CASE Testausdruck
CASE Ausdrucksliste1
  [Anweisungsblock-1]
[CASE Ausdrucksliste2
  [Anweisungsblock-2]]
.
.
.
[CASE ELSE
  [Anweisungsblock-n]]
END SELECT
```

SETMEM-Funktion

Verändert die Größe des Speichers, der vom Far Heap genutzt wird - der Bereich, in dem weite Objekte (Far Objects) und interne Tabellen gespeichert werden.

Syntax **SETMEM** (*Numerischer Ausdruck*)

SGN-Funktion

Gibt das Vorzeichen eines numerischen Ausdrucks an.

Syntax **SGN** (*Numerischer Ausdruck*)

SHARED-Anweisung

Ermöglicht einer **SUB**- oder **FUNCTION**-Prozedur Zugriff auf im Modul-Ebenen-Code deklarierte Variablen, ohne diese als Parameter zu übergeben.

Syntax **SHARED** *Variable* [**AS***Typ*][,*Variable* [**AS***Typ*]]...

SHELL-Anweisung

Verläßt das BASIC-Programm, führt ein *.COM*, *.exe* oder *.bat*-Programm oder einen DOS-Befehl aus und kehrt zum Programm in die Zeile nach der Anweisung **SHELL** zurück.

Syntax **SHELL** [*Befehlszeichenkette*]

SIN-Funktion

Gibt den Sinus des in Bogenmaß angegebenen Winkels *x* an.

Syntax **SIN**(*x*)

SLEEP-Anweisung

Setzt die Ausführung des aufrufenden Programmes aus.

Syntax **SLEEP** [*Sekunden*]

SOUND-Anweisung

Erzeugt Töne über den Lautsprecher.

Syntax **SOUND** *Frequenz*, *Länge*

SPACE$-Funktion

Liefert eine Zeichenkette von *n* Leerzeichen.

Syntax **SPACE$**(*n*)

SPC-Funktion

Überspringt in einer **PRINT**-Anweisung *n* Leerzeichen.

Syntax **SPC**(*n*)

SQR-Funktion

Gibt die Quadratwurzel von *n* an.

Syntax **SQR**(*n*)

STATIC-Anweisung

Macht einfache Variablen oder Datenfelder lokal zu einer **DEF FN**-Funktion, **FUNCTION** oder **SUB** und speichert die Werte zwischen Aufrufen.

Syntax **STATIC** *Variablenliste*

STICK-Funktion

Gibt die *x*- und *y*-Koordinaten der beiden Joysticks an.

Syntax **STICK**(*n*)

STOP-Anweisung

Beendet das Programm.

Syntax **STOP**

STR$-Funktion

Liefert eine Zeichenkettendarstellung des Wertes eines numerischen Ausdrucks.

Syntax **STR$** (*Numerischer Ausdruck*)

STRIG-Funktion und -Anweisung

Liefert den Status des angegebenen Joysticks.

Syntax 1 (FUNCTION) **STRIG(***n***)**

Syntax 2 (ANWEISUNG) **STRIG {ON | OFF}**

STRIG ON-, OFF- UND STOP-Anweisungen

Aktivieren, deaktivieren oder sperren die Verfolgung der Joysticktätigkeit.

Syntax **STRIG(***n***) ON**
STRIG(*n***) OFF**
STRIG(*n***) STOP**

STRING$-Funktion

Gibt eine Zeichenkette zurück, deren Zeichen alle einen gegebenen ASCII-Code haben oder deren Zeichen alle das erste Zeichen eines Zeichenkettenausdruckes darstellen.

Syntax **STRING$** (*m*, *n*)
STRING$ (*m*, *Zeichenkettenausdruck*)

SUB-Anweisungen

Kennzeichnen Anfang und Ende eines Unterprogrammes.

Syntax **SUB** *Globalname* [(*Parameterliste*)]**[STATIC]**

.
.
.

[EXIT SUB]

.
.
.

END SUB

SWAP-Anweisung

Tauscht die Werte zweier Variablen aus.

Syntax **SWAP** *Variable1*, *Variable2*

SYSTEM-Anweisung

Schließt alle geöffneten Dateien und gibt die Kontrolle an das Betriebssystem zurück.

Syntax **SYSTEM**

TAB-Funktion

Bewegt die Ausgabeposition.

Syntax **TAB** (*Spalte*)

TAN-Funktion

Die TAN-Funktion (Tangens) gibt den Tangens des im Bogenmaß angegebenen Winkels *x* an.

Syntax **TAN**(*x*)

TIME$-Funktion

Gibt die aktuelle Uhrzeit des Betriebssystems an.

Syntax **TIME$**

TIME$-Anweisung

Setzt die Zeit.

Syntax **TIME$** = *Zeichenkettenausdruck*

TIMER-Funktion

Gibt die Anzahl der seit Mitternacht vergangenen Sekunden an.

Syntax **TIMER**

TIMER ON-, OFF- und STOP-Anweisungen

Aktiviert, deaktiviert oder sperrt die Ereignisverfolgung des Zeitgebers.

Syntax **TIMER ON**
TIMER OFF
TIMER STOP

TRON/TROFF-Anweisungen

Verfolgen die Ausführung von Programmanweisungen.

Syntax **TRON**
TROFF

TYPE-Anweisung

Definiert einen Datentyp, der ein oder mehrere Elemente enthält.

Syntax **TYPE** *Benutzertyp*
Elementname **AS** *Typname*
Elementname **AS** *Typname*
.
.
.
END TYPE

UBOUND-Funktion

Ermittelt die obere Grenze (den größten vorhandenen Index) für die angegebene Dimension eines Datenfeldes.

Syntax **UBOUND** *(Datenfeld* [, *Dimension*])

UCASE$-Funktion

Gibt einen Zeichenkettenausdruck aus, in dem alle Buchstaben groß geschrieben sind.

Syntax **UCASE$** (*Zeichenkettenausdruck*)

UEVENT-Anweisung

Aktiviert, deaktiviert oder sperrt die Ereignisverfolgung für ein benutzerdefiniertes Ereignis.

Syntax **UEVENT ON**
UEVENT OFF
UEVENT STOP

UNLOCK-Anweisung

Hebt die für Teile einer Datei geltenden Zugriffsbeschränkungen auf.

Syntax **UNLOCK** [#] *Dateinummer* [,{*Datensatz* | [*Anfang*] **TO** *Ende*}]

VAL-Funktion

Gibt den numerischen Wert einer Zeichenkette von Ziffern an.

Syntax **VAL** (*Zeichenkettenausdruck*)

VARPTR-, VARSEG-Funktionen

Geben die Adressen einer Variablen an.

Syntax **VARPTR** (*Variablenname*)
VARSEG (*Variablenname*)

VARPTR$-Funktion

Gibt die Zeichenkettendarstellung einer Variablenadresse zur Verwendung in den Anweisungen **DRAW** und **PLAY** zurück.

Syntax **VARPTR$** (Variablenname)

VIEW-Anweisung

Definiert die Bildschirmgrenzen für die graphische Ausgabe.

Syntax **VIEW** [[**SCREEN**] (*x1,y1*) - (*x2,y2*) [,[*Farbe*][, *Rand*]]]

VIEW PRINT-Anweisung

Setzt die Grenzen des Bildschirm-Textfensters.

Syntax **VIEW PRINT** [*Kopfzeile* **TO** *Fußzeile*]

WAIT-Anweisung

Unterbricht die Programmausführung während der Status eines Maschinen-Eingabeanschlusses (Input Port) überwacht wird.

Syntax **WAIT** *Anschlußnummer*, *UND-Ausdruck* [, *XOR-Ausdruck*]

WHILE...WEND-Anweisung

Führt eine Folge von Anweisungen in einer Schleife aus, solange eine gegebene Bedingung wahr ist.

Syntax **WHILE** *Bedingung*

.
.
.
[*Anweisungen*]
.
.
.
WEND

WIDTH-Anweisung

Weist einer Datei oder einem Gerät die Breite der Ausgabezeile zu, oder ändert die auf dem Bildschirm angezeigte Anzahl von Spalten und Zeilen.

Syntax **WIDTH** [*Spalten*][, *Zeilen*]
WIDTH {*#Dateinummer* | *Gerät*}, *Breite*
WIDTH LPRINT *Breite*

WINDOW-Anweisung

Definiert die logischen Dimensionen des laufenden Darstellungsfeldes.

Syntax **WINDOW** [[**SCREEN**] (*x1*,*y1*) - (*x2*,*y2*)]

WRITE-Anweisung

Sendet Daten an den Bildschirm.

Syntax **WRITE** [*Ausdrucksliste*]

WRITE#-Anweisung

Schreibt Daten in eine sequentielle Datei.

Syntax **WRITE** # *Dateinummer*, *Ausdrucksliste*

9 Quick-Referenztabellen

Alle Abschnitte dieses Kapitels fassen eine Gruppe von Anweisungen oder Funktionen zusammen, die typischerweise zusammen verwendet werden. Jede Gruppe wird in einer Tabelle vorgestellt, die den Typ der ausgeführten Aufgabe (beispielsweise Durchführung von Schleifen oder Suchvorgängen), den Namen der Anweisung oder Funktion sowie die Aktion der Anweisung aufführt.

Folgende Anweisungen und Funktionen werden tabellenförmig zusammengefaßt:

- Anweisungen zur Ablaufsteuerung.
- In BASIC-Prozeduren verwendete Anweisungen.
- Standard-E/A-Anweisungen.
- Datei-E/A-Anweisungen.
- Anweisungen und Funktionen für die Zeichenkettenverarbeitung.
- Graphik-Anweisungen und -Funktionen.
- Verfolgungs-Anweisungen und -Funktionen.

Diese Tabellen können als Referenz zu der Aktion jeder Anweisung und Funktion sowie zur Identifizierung zusammenhängender Anweisungen dienen.

9.1 Zusammenfassung der Anweisungen zur Ablaufsteuerung

Tabelle 9.1 führt die in BASIC zur Programmablaufsteuerung verwendeten Anweisungen auf.

Tabelle 9.1 In der Durchführung von Schleifen und Fällen von Entscheidungen verwendete Anweisungen

Aufgabe	***Anweisung***	***Funktion***
Schleife	**FOR...NEXT**	Wiederholt Anweisungen zwischen **FOR** und **NEXT** sooft wie angegeben.
	EXIT FOR	Bietet eine Alternative zum Verlassen einer **FOR...NEXT**-Schleife.
	DO LOOP	Wiederholt Anweisungen zwischen **DO** und **LOOP** solange eine gegebene Bedingung wahr ist (**DO...LOOP WHILE** *Bedingung*) oder bis sie wahr wird (**DO...LOOP UNTIL** *Bedingung*).
	EXIT DO	Bietet einen weiteren Weg, eine **DO...LOOP**-Schleife zu verlassen.
	WHILE...WEND	Wiederholt Anweisungen zwischen **WHILE** und **WEND** solange eine gegebene Bedingung wahr ist (ähnlich wie **DO WHILE** *Bedingung***...LOOP**).
Fällen von Entscheidungen	**IF...THEN...ELSE**	Verzweigt bedingt oder führt verschiedene Anweisungen aus.
	SELECT CASE	Führt verschiedene Anweisungen bedingt aus.

9.2 Zusammenfassung der in BASIC-Prozeduren verwendeten Anweisungen

Tabelle 9.2 führt die in BASIC benutzten Anweisungen auf, die BASIC-Prozeduren definieren, deklarieren, aufrufen und Argumente an sie übergeben. Die Tabelle listet ebenfalls Anweisungen auf, die zum gemeinsamen Gebrauch von Variablen zwischen Prozeduren, Modulen und separaten Programmen in einer Kette verwendet werden.

Tabelle 9.2 In Prozeduren verwendete Anweisungen

Aufgabe	*Anweisung*	*Funktion*
Definieren einer Prozedur	**FUNCTION... ENDFUNCTION**	Markiert den Beginn bzw. das Ende einer **FUNCTION**-Prozedur.
	SUB...END SUB	Markiert den Beginn bzw. das Ende einer **SUB**-Prozedur.
Aufruf einer Prozedur	**CALL**	Übergibt die Kontrolle an eine BASIC-**SUB**-Prozedur oder an eine in einer anderen Programmiersprache geschriebene und getrennt kompilierte Prozedur. (Das Schlüsselwort **CALL** ist wahlfrei).
Verlassen einer Prozedur	**EXIT FUNCTION**	Bietet eine alternative Möglichkeit, eine **FUNCTION**-Prozedur zu verlassen.
	EXIT SUB	Bietet eine alternative Möglichkeit, eine **SUB**-Prozedur zu verlassen.
Bezugnahme auf eine Prozedur vor deren Definierung	**DECLARE**	Deklariert eine **FUNCTION** oder **SUB** und gibt optional die Anzahl und Typen ihrer Parameter an.

Fortsetzung auf der folgenden Seite.

Aufgabe	*Anweisung*	*Funktion*
Variablen zwischen Modulen, Prozeduren oder Programmen	**COMMON**	Ermöglicht gemeinsame Nutzung von Variablen zwischen einzelnen Modulen. Bei der gleichzeitigen Verwendung des Attributs **SHARED** werden Variablen zwischen unterschiedlichen Prozeduren des gleichen Moduls gemeinsam benutzt. Übergibt auch Variablenwerte vom laufenden Programm in ein neues Programm, wenn die Kontrolle mit Hilfe der Anweisung **CHAIN** übertragen wird.
	SHARED	Wird sie mit den Anweisungen **COMMON, DIM** oder **REDIM** auf Modul-Ebene (zum Beispiel **DIM SHARED**) verwendet, werden Variablen jeder **SUB** oder **FUNCTION** in einem einzelnen Modul gemeinsam benutzt. Wird sie separat innerhalb einer Prozedur verwendet, werden Variablen zwischen dieser Prozedur und dem Modul-Ebenen-Code gemeinsam benutzt.
Erhaltung der Variablenwerte	**STATIC**	Zwingt Variablen, lokal zu einer Prozedur oder **DEF FN**-Funktion zu sein, und behält die in den Variablen gespeicherten Werte bei, wenn die Prozedur oder Funktion verlassen und anschließend wieder aufgerufen wird.
Definieren einer mehrzeiligen Funktion	**DEF FN...** **END DEF**	Markiert den Beginn und das Ende einer mehrzeiligen **DEF FN**-Funktion (es handelt sich um die alte Version der BASIC-Funktionen; **FUNCTION**-Prozeduren bieten eine leistungsfähige Alternative).

Aufgabe	*Anweisung*	*Funktion*
Verlassen einer mehrzeiligen Funktion	**EXIT DEF**	Bietet eine Alternative, eine mehrzeilige **DEF FN**-Funktion zu verlassen.
Aufruf einer BASIC-Unterroutine	**GOSUB**	Übergibt die Kontrolle an eine bestimmte Zeile in einem Modul. Die Kontrolle wird von der Unterroutine anhand einer **RETURN**-Anweisung an die Zeile zurückgegeben, die der **GOSUB**-Anweisung folgt (dies ist die alte Version der BASIC-Unterroutinen; **SUB**-Prozeduren bieten eine leistungsfähige Alternative).
Übergabe der Kontrolle an ein anderes Programm	**CHAIN**	Überträgt die Kontrolle vom laufenden Programm im Speicher an ein anderes Programm; zur Übergabe von Variablen ist **COMMON** zu verwenden.

9.3 Zusammenfassung der Standard-E/A-Anweisungen

Tabelle 9.3 führt die in BASIC für Standard-E/A (typischerweise Eingabe von der Tastatur und Ausgabe auf den Bildschirm) verwendeten Anweisungen und Funktionen auf.

Tabelle 9.3 Für Standard-E/A verwendete Anweisungen und Funktionen

Aufgabe	*Anweisung oder Funktion*	*Funktion*
Textausgabe auf dem Bildschirm	**PRINT**	Gibt Text auf den Bildschirm aus. Wird **PRINT** ohne Argumente benutzt, wird eine leere Zeile ausgegeben.
	PRINT USING	Gibt formatierten Text auf den Bildschirm aus.
Verändern der Breite einer Ausgabezeile	**WIDTH**	Verändert die Breite des Bildschirms auf 40 oder 80 Spalten und steuert auf Computern mit einer EGA- oder VGA-Karte die Zeilenanzahl auf dem Bildschirm (25 oder 43).
	WIDTH "SCRN:"	Weist vor einer **OPEN "SCRN:"** Anweisung verwendeten Zeilen eine maximale Länge auf dem Bildschirm zu.
Lesen der Tastatureingabe	**INKEY$**	Liest ein Zeichen von der Tastatur (oder eine Nullzeichenkette(""), falls kein Zeichen in der Warteschlange vorhanden ist).
	INPUT$	Liest eine bestimmte Zeichenanzahl von der Tastatur und speichert diese in einer einzigen Zeichenkettenvariablen.
	INPUT	Liest Eingabe von der Tastatur und speichert diese in einer Variablenliste.

Aufgabe	*Anweisung oder Funktion*	*Funktion*
	LINE INPUT	Liest eine Eingabezeile von der Tastatur und speichert sie in einer einzigen Zeichenkettenvariablen.
Positionieren des Cursors auf dem	**LOCATE**	Bewegt den Cursor zu einer gegebenen Zeile und Spalte.
Bildschirm	**SPC**	Überspringt Leerzeichen in einer Ausgabezeile.
	TAB	Zeigt Ausgabe in einer gegebenen Spalte an.
Informationen über die Position des Cursors	**CSRLIN**	Gibt an, in welcher Zeile der Cursor steht.
	POS(*n*)	Gibt die Spalte an, in der sich der Cursor befindet.
Erstellen eines Text-Darstel-lungsfeldes	**VIEW PRINT**	Setzt die oberste und unterste Zeile zur Anzeige der Textausgabe fest.

9.4 Zusammenfassung der Datei-E/A-Anweisungen

Tabelle 9.4 listet die in der BASIC-Datendatei-Programmierung benutzten Anweisungen und Funktionen auf.

Tabelle 9.4 In der Datendatei-E/A verwendete Anweisungen und Funktionen

Aufgabe	*Anweisung oder Funktion*	*Funktion*
Erstellung einer neuen Datei oder Zugriff auf eine vorhandene Datei	**OPEN**	Öffnet eine Datei zum Lesen oder Speichern von Datensätzen (E/A).
Schließen einer Datei	**CLOSE**	Beendet E/A-Operationen in einer Datei.
Speichern von Daten in einer Datei	**PRINT#**	Speichert Variablen als Datensatzfelder in einer zuvor geöffneten Datei.*
	PRINT USING#	Ähnlich wie **PRINT#**, mit dem Unterschied, daß **PRINT USING#** die Datensatzfelder formatiert.*
	WRITE#	Speichert eine Variablenliste als Datensatzfelder in einer zuvor geöffneten Datei.*
	WIDTH	Legt eine Standardlänge für jeden Datensatz einer Datei fest.*
	PUT	Speichert den Inhalt einer benutzerdefinierten Variablen in einer zuvor geöffneten Datei.**
Lesen von Daten aus einer Datei	**INPUT#**	Liest Felder eines Datensatzes und weist jedes Datensatzfeld einer Programmvariablen zu.*

Aufgabe	*Anweisung oder Funktion*	*Funktion*
	INPUT$	Liest eine Zeichenkette aus einer Datei.
	LINE INPUT#	Liest einen Datensatz und speichert diesen in einer einzigen Zeichenkettenvariablen.*
	GET	Liest Daten aus einer Datei und weist diese den Elementen einer benutzerdefinierten Variablen zu.**
Verwaltung der Dateien auf der Diskette	**FILES**	Gibt die Dateiliste eines angegebenen Verzeichnisses aus.
	FREEFILE	Gibt die nächste verfügbare Dateinummer an.
	KILL	Löscht eine Datei auf der Diskette.
	NAME	Verändert den Namen einer Datei.
Informationen über eine Datei	**EOF**	Prüft, ob alle Daten einer Datei gelesen wurden.
	FILEATTR	Gibt die vom Betriebssystem einer geöffneten Datei zugewiesene Nummer und die Nummer, die den Modus der geöffneten Datei anzeigt, zurück (**INPUT**, **OUTPUT**, **APPEND**, **BINARY** oder **RANDOM**).
	LOC	Gibt die aktuelle Position innerhalb einer Datei an. Bei binärem Zugriff handelt es sich um die Byte-Position. Bei sequentiellem Zugriff ist es die durch 128 dividierte Byte-Position. Bei Direktzugriff ist es die Datensatznummer des letzten gelesenen oder geschriebenen Datensatzes.

Fortsetzung auf der folgenden Seite.

Aufgabe	*Anweisung oder Funktion*	*Funktion*
	LOF	Gibt die Byteanzahl einer geöffneten Datei an.
	SEEK(Funktion)	Gibt die Position der nächsten E/A-Operation an. Bei Direktzugriff handelt es sich um die Nummer des nächsten zu lesenden oder zu schreibenden Datensatzes. Bei allen anderen Arten des Dateizugriffes geht es um die Byteposition des nächsten zu lesenden oder zu schreibenden Bytes.
Bewegen in einer Datei	**SEEK** (Anweisung)	Setzt die Byteposition des nächsten Lese- oder Schreibvorganges in einer geöffneten Datei.

* Mit sequentiellen Dateien zu verwenden.

** Mit Binär- oder Direktzugriffsdateien zu verwenden.

9.5 Zusammenfassung der Anweisungen und Funktionen zur Zeichenkettenverarbeitung

Tabelle 9.5 führt die in BASIC bei der Arbeit mit Zeichenketten zur Verfügung stehenden Anweisungen und Funktionen auf.

Tabelle 9.5 Für Zeichenkettenverarbeitung verwendete Anweisungen und Funktionen

Aufgabe	***Anweisung oder Funktion***	***Funktion***
Separieren eines Zeichenket-tenteiles	**LEFT$**	Gibt eine gegebene Anzahl von Zeichen von der linken Seite einer Zeichenkette zurück.
	RIGHT$	Gibt eine gegebene Zeichenanzahl von der rechten Seite einer Zeichenkette zurück.
	LTRIM$	Gibt die Kopie einer Zeichenkette zurück, deren führende Leerzeichen entfernt wurden.
	RTRIM$	Gibt die Kopie einer Zeichenkette zurück, deren nachfolgende Leerzeichen entfernt wurden.
	MID$ (Funktion)	Gibt eine gegebene Anzahl von Zeichen ab einer beliebigen Stelle in der Zeichenkette zurück.
Suchen nach einer Zeichenkette	**INSTR**	Sucht nach einer Zeichenkette innerhalb einer anderen Zeichenkette.
Umwandeln von Klein in Großbuchstaben bzw. umgekehrt	**LCASE$**	Gibt die Kopie einer Zeichenkette zurück, in der alle Großbuchstaben (A - Z) in Kleinbuchstaben (a - z) umgewandelt wurden; Kleinbuchstaben und andere Zeichen bleiben unverändert.

Fortsetzung auf der folgenden Seite.

Aufgabe	*Anweisung oder Funktion*	*Funktion*
	UCASE$	Gibt die Kopie einer Zeichenkette zurück, in der alle Kleinbuchstaben (a - z) in Großbuchstaben (A - Z) umgewandelt wurden; Großbuchstaben und andere Zeichen bleiben unverändert.
Verändern der Zeichenketten	**MID$** (Anweisung)	Ersetzt einen Zeichenkettenteil mit einer anderen Zeichenkette.
	LSET	Ordnet eine Zeichenkette innerhalb einer Zeichenkette fester Länge linksbündig an.
	RSET	Ordnet eine Zeichenkette innerhalb einer Zeichenkette fester Länge rechtsbündig an.
Umwandeln von Zahlen in Zeichenketten und umgekehrt	**STR$**	Gibt die Zeichenkettendarstellung eines numerischen Ausdruckswertes zurück.
	VAL	Gibt den numerischen Wert eines Zeichenkettenausdruckes zurück.
Umwandeln von Zahlen in Datendateizeichenketten* und umgekehrt	**CV***Typ*	Wandelt in Programmen, die mit in älteren BASIC-Versionen erstellten Direktzugriffsdateien arbeiten, als Zeichenketten gespeicherte Zahlen in Microsoft Binärformatzahlen um.
	CV*Typ***MBF**	Wandelt die als Zeichenketten im Microsoft Binärformat gespeicherten Zahlen in IEEE-Formatzahlen um.
	MK*Typ***$**	Wandelt Microsoft Binärformatzahlen in Zeichenketten um, die zur Speicherung in Direktzugriffsdateien dienen, die mit älteren BASIC-Versionen erstellt wurden.

Aufgabe	*Anweisung oder Funktion*	*Funktion*
	MK*Typ***MBF$**	Wandelt IEEE-Formatzahlen in Zeichenketten im Microsoft Binärformat um.
Bilden von Zeichenketten mit sich wiederholenden Zeichen	**SPACE$**	Gibt eine aus Leerzeichen bestehende Zeichenkette bestimmter Länge zurück.
	STRING$	Gibt eine aus einem einzigen wiederholten Zeichen bestehende Zeichenkette zurück.
Feststellen der Länge einer Zeichenkette	**LEN**	Gibt die Zeichenanzahl in einer Zeichenkette an.
Arbeiten mit ASCII-Werten	**ASC**	Gibt den ASCII-Wert des angegebenen Zeichens zurück.
	CHR$	Gibt das Zeichen mit dem angegebenen ASCII-Wert zurück.

* Dies ist für Datensätze im Direktzugriff, deren Datenstruktur mit **TYPE...END TYPE** definiert ist, nicht länger erforderlich.

9.6 Zusammenfassung der Graphikanweisungen und -funktionen

Tabelle 9.6 führt alle Anweisungen und Funktionen auf, die in BASIC für auf Bildpunkte aufbauende Graphik verwendet werden.

Tabelle 9.6 Für Graphikausgabe verwendete Anweisungen und Funktionen

Aufgabe	*Anweisung oder Funktion*	*Funktion*
Setzen der Bildschirmeigenschaften	**SCREEN**	Legt einen BASIC-Bildschirmmodus fest, der Bildschirmeigenschaften, wie Auflösung und Bereiche für Farbnummern, bestimmt.
Zeichnen oder Löschen eines einzelnen Punktes	**PSET**	Gibt einem Bildpunkt auf dem Bildschirm eine bestimmte Farbe; verwendet standardmäßig die Vordergrundfarbe des Bildschirms.
	PRESET	Gibt einem Bildpunkt auf dem Bildschirm eine bestimmte Farbe; verwendet standardmäßig die Hintergrundfarbe des Bildschirms, löscht damit effektiv den Bildpunkt.
Zeichnen von Formen	**LINE**	Zeichnet eine gerade Linie oder ein Rechteck.
	CIRCLE	Zeichnet einen Kreis oder eine Ellipse.
	DRAW	Kombiniert viele Eigenschaften anderer BASIC-Graphikanweisungen (Zeichnen von Geraden, Bewegen des graphischen Cursors, Skalieren von Bildern) in einer "all-in-one" graphischen Makro-Sprache.

Aufgabe	*Anweisung oder Funktion*	*Funktion*
Definieren von Bildschirmkoordinaten	**VIEW**	Legt ein Rechteck auf dem Bildschirm (oder Darstellungsfeld) als Bereich für die graphische Ausgabe fest.
	WINDOW	Ermöglicht es dem Benutzer, neue logische Koordinaten für ein Darstellungsfeld auf dem Bildschirm zu wählen.
	PMAP	Bildet physikalische Bildschirmkoordinaten auf vom Benutzer im aktuellen Fenster festgelegte logische Koordinaten ab und umgekehrt.
	POINT(*Zahl*)	Gibt die aktuellen physikalischen oder logischen Koordinaten des graphischen Cursors je nach dem Wert für *Zahl* an.
Verwenden von Farben	**COLOR**	Setzt die in der graphischen Ausgabe standardmäßig verwendeten Farben.
	PALETTE	Weist Farbnummern andere Farben zu. Funktioniert nur auf Systemen, die mit einer EGA- oder VGA-Graphikkarte ausgerüstet sind.
	POINT(*x*,*y*)	Gibt die Farbnummer eines einzelnen Bildpunktes mit den Bildschirmkoordinaten *x* und *y* an.
Ausmalen geschlossener Formen	**PAINT**	Füllt einen Bildschirmbereich mit einer Farbe oder einem Muster aus.
Animation	**GET**	Kopiert einen rechteckigen Bereich des Bildschirms durch Übertragen des Bildes in numerische Daten und Speichern dieser Daten in einem numerischen Datenfeld.
	PUT	Gibt ein mit **GET** kopiertes Bild auf dem Bildschirm aus.
	PCOPY	Kopiert eine Bildschirmseite in eine andere.

9.7 Zusammenfassung der Anweisungen und Funktionen zur Fehler- und Ereignisverfolgung

Tabelle 9.7 listet die Anweisungen und Funktionen auf, die von BASIC zur Verfolgung und Bearbeitung von Fehlern und Ereignissen verwendet werden.

Tabelle 9.7 Zur Fehler- und Ereignisverfolgung verwendete Anweisungen und Funktionen

Aufgabe	*Anweisung oder Funktion*	*Funktion*
Fehlerverfolgung während des Programmablaufes	**ON ERROR GOTO** *Zeile*	Veranlaßt ein Programm, in die angegebene *Zeile* zu verzweigen, wobei *Zeile* sich auf eine Zeilennummer oder Zeilenmarke bezieht. Die Verzweigung findet während der Ausführung des Programmes jedesmal statt, wenn ein Fehler auftritt.
	RESUME	Gibt die Kontrolle nach Ausführung einer Fehlerbehandlungsroutine an das Programm zurück. Das Programm fährt entweder mit der fehlerverursachenden Anweisung (**RESUME** [**0**]), der darauffolgenden Anweisung (**RESUME NEXT**) oder der von *Zeile* angegebenen Zeile (**RESUME** *Zeile*) fort.
Lesen von Fehlerstatusdaten	**ERR**	Gibt den Code für einen Laufzeitfehler zurück.
	ERL	Gibt Zeilennummer an, in der der Fehler aufgetreten ist (wenn das Programm über Zeilennummern verfügt).
	ERDEV	Gibt einen gerätespezifischen Fehlercode für das letzte Gerät (wie einen Drucker) an, auf dem DOS einen Fehler entdeckt hat.

Aufgabe	*Anweisung oder Funktion*	*Funktion*
	ERDEV$	Gibt den Namen des letzten Gerätes zurück, auf dem DOS einen Fehler entdeckt hat.
Definieren des eigenen Fehlercodes	**ERROR**	Simuliert das Auftreten eines BASIC-Fehlers; kann auch zum Definieren eines nicht von BASIC verfolgten Fehlers verwendet werden.
Ereignisverfolgung während des Programmablaufes	**ON** *Ereignis* **GOSUB** *Zeile*	Verursacht eine Verzweigung in die mit *Zeile* beginnende Unterroutine, wobei *Zeile* sich entweder auf eine Zeilennummer oder eine Zeilenmarke bezieht. Die Verzweigung wird beim Auftreten des angegebenen *Ereignisses* während des Ablaufes ausgeführt.
	Ereignis **ON**	Schaltet die Verfolgung des angegebenen *Ereignisses* ein.
	Ereignis **OFF**	Schaltet die Verfolgung des angegebenen *Ereignisses* aus.
	Ereignis **STOP**	Setzt die Verfolgung des angegebenen *Ereignisses* aus.
	RETURN	Gibt die Kontrolle nach Ausführung einer Ereignisbehandlungsroutine an das Programm zurück. Das Programm fährt entweder mit der Anweisung fort, die im Programm unmittelbar nach der Stelle steht, an der das Ereignis aufgetreten ist (**RETURN**), oder mit der in *Zeile* angegebenen Zeile (**RETURN** *Zeile*).

Anhangsverzeichnis

A Übertragen von BASICA-Programmen in QuickBASIC
B Unterschiede zu früheren QuickBASIC-Versionen
C Einschränkungen für QuickBASIC
D Tastaturabfragecodes und ASCII-Zeichencodes
E Reservierte Wörter in BASIC
F Metabefehle
G Kompilieren und Binden aus DOS
H Erstellung und Verwendung von Quick-Bibliotheken
I Fehlermeldungen

Anhang A: Übertragen von BASICA-Programmen in QuickBASIC

QuickBASIC akzeptiert im allgemeinen für BASIC-Interpreter in BASIC geschriebene Anweisungen, die auf IBM Personal Computern und Kompatiblen eingesetzt werden: IBM Advanced Personal Computer BASIC Version A3.00 (BASICA) und Microsoft® GW-BASIC®. Einige Änderungen sind jedoch aufgrund interner Unterschiede zwischen QuickBASIC und diesen BASIC-Interpretern notwendig. Da die Änderungen für beide Interpreter gelten, bezieht sich in diesem Anhang der Begriff BASICA sowohl auf GW-BASIC als auch auf BASICA.

Dieser Anhang erteilt Informationen zu folgenden Punkten:

- Kompatibles Format einer Quelldatei.
- Anweisungen und Funktionen, die nicht zulässig sind bzw. Änderungen erfordern, um in QuickBasic benutzt werden zu können.
- Editorunterschiede in der Handhabung von Tabulatoren.

Die folgenden Abschnitte beschreiben nur die erforderlichen Änderungen, um ein BASICA-Programm mit der QuickBASIC Version 4.5 zu kompilieren und zu starten. Diese Kapitel enthalten Informationen, wie Sie mit Hilfe der QuickBASIC-Eigenschaften Verbesserungen an vorhandenen BASICA-Programmen vornehmen.

A.1 Format einer Quelldatei

QuickBASIC Version 4.5 erwartet, daß die Quelldatei im ASCII-Format oder im QuickBASIC-Format vorliegt. Eine mit BASICA erstellte Datei sollte mit Hilfe der Option ,A abgespeichert werden; andernfalls komprimiert BASICA den Text des Programmes in einem besonderen, für QuickBASIC unlesbaren Format. In diesem Fall ist BASICA erneut zu laden und die Datei mit Hilfe der Option ,A im ASCII-Format abzuspeichern. Zum Beispiel speichert der folgende BASICA-Befehl die Datei *meinprog.bas* im ASCII-Format ab:

```
SAVE "meinprog.bas",A
```

A.2 In QuickBASIC nicht zulässige Anweisungen und Funktionen

Die unten aufgeführten Anweisungen und Funktionen dürfen nicht in einem QuickBASIC-Programm verwendet werden, da sie eine Quelldatei bearbeiten, in die Programmausführung eingreifen, sich auf Kassetten-Geräte beziehen, die von QuickBASIC nicht unterstützt werden, oder mit einer in der QuickBASIC-Umgebung bereits bereitgestellten Hilfe übereinstimmen.

AUTO	**LIST**	**NEW**
CONT	**LLIST**	**RENUM**
DEF USR	**LOAD**	**SAVE**
DELETE	**MERGE**	**USR**
EDIT	**MOTOR**	

A.3 Anweisungen, die Änderungen erfordern

Falls das BASICA-Programm eine der in Tabelle A.1 aufgeführten Anweisungen enthält, müssen Sie den Quellcode vor Kompilieren und Starten des Programms in QuickBASIC wahrscheinlich modifizieren.

Tabelle A.1 Anweisungen die Änderungen erfordern

Anweisung	***Änderung***
BLOAD/ BSAVE	Speicherstellen können in QuickBASIC verschieden sein.
CALL	Das Argument ist der Name der aufgerufenen **SUB**-Prozedur.
CHAIN	QuickBASIC unterstützt nicht die Optionen **ALL, MERGE, DELETE** oder *Zeilennummer*.
COMMON	**COMMON**-Anweisungen müssen vor jeglichen ausführbaren Anweisungen erscheinen.
DEF*Typ*	**DEF***Typ*-Anweisungen sollten an den Anfang der BASICA-Quelldatei übertragen werden.
DIM	Alle **DIM**-Anweisungen, welche statische Datenfelder deklarieren, müssen am Anfang von QuickBASIC-Programmen erscheinen.
DRAW, PLAY	QuickBASIC erfordert die Verwendung der Funktion **VARPTR$** mit eingebetteten Variablen.

Anweisung	*Änderung*
RESUME	Beim Auftreten eines Fehlers in einer einzeiligen Funktion versucht QuickBASIC die Programmausführung mit der Zeile fortzusetzen, die die Funktion enthält.
RUN	Für ausführbare, mit QuickBASIC erzeugte Dateien darf das Objekt einer **RUN**- oder **CHAIN**-Anweisung keine *.bas*-Datei, sondern nur eine ausführbare Datei sein. Die BASICA-Option R wird nicht unterstützt. Während es in der QuickBASIC-Umgebung ausgeführt wird, ist das Objekt einer **RUN**- oder **CHAIN**-Anweisung immer noch eine *.bas*-Datei. **RUN** {*Zeilennummer* | *Zeilenmarke*} wird jedoch von QuickBASIC unterstützt; diese Anweisung startet das Programm ab der angegebenen Zeile neu.

A.4 Editorunterschiede in der Handhabung von Tabulatoren

QuickBASIC benutzt einzelne Leerzeichen, nicht das literale Tabulatorzeichen ASCII 9 zur Darstellung der Einrückungsebenen. Wählen Sie die Option **Tab-Abstand** im Dialogfeld **Bildschirm** des Menüs **Optionen**, um die Leerzeichenanzahl der jeweiligen Einrückungsebenen festzusetzen. Nähere Informationen entnehmen Sie bitte Kapitel 20, "Das Menü Optionen" des Handbuches *Lernen und Anwenden von Microsoft QuickBASIC*.

Einige Texteditoren verwenden literale Tabulatorzeichen zur Darstellung mehrerer Leerzeichen beim Speichern von Textdateien. Der QuickBASIC-Editor behandelt literale Tabulatorzeichen in solchen Dateien wie folgt:

1. Literale Tabulatorzeichen innerhalb einer Zeichenkette in doppelten Anführungszeichen erscheinen wie das Zeichen, das in der ASCII-Tabelle im Anhang D, "Tastaturabfragecodes und ASCII-Zeichencodes", unter Nummer 9 aufgeführt ist.
2. Außerhalb von Zeichenketten in doppelten Anführungszeichen richten Tabulatorzeichen den darauffolgenden Text auf die nächste Einrückungsebene aus.

Beim Versuch, die Tabulatorstoppeinstellung zu ändern, während ein Programm dieser Art geladen ist, gibt QuickBASIC folgende Fehlermeldung aus:

```
Tab-Stopps können nicht geändert werden, während diese Datei
geladen ist
```

Diese Fehlermeldung verhindert die versehentliche Neuformatierung alter, anhand anderer Editoren erstellter Quelldateien. Wenn Sie sich nach dem Laden einer Datei dieser Art entscheiden, anders einzurücken, ist die Einrückung wie folgt neu einzustellen:

1. Speichern Sie die Datei, um sämtliche Änderungen beizubehalten, und wählen Sie anschließend den Befehl **Neues Programm** aus dem Menü **Datei.**
2. Wählen Sie den Befehl **Bildschirm** aus dem Menü **Optionen** und setzen Sie die Option Tab-Abstand wie oben fest.
3. Wählen Sie den Befehl **Programm laden** aus dem Menü **Datei** und laden Sie das Programm erneut. Nach Laden des Programms werden die Einrückungen die neue Einstellung der Option **Tab-Abstand** wiedergeben.

Es ist dabei zu beachten, daß dieses Verfahren nur für alte Programme gilt. Mit QuickBASIC erstellter Text läßt sich nicht auf diese Weise neu formatieren.

Anhang B: Unterschiede zu früheren QuickBASIC-Versionen

QuickBASIC Version 4.5 enthält zahlreiche neue Eigenschaften und Verbesserungen gegenüber vorhergehenden QuickBASIC-Versionen. Dieser Anhang beschreibt die Unterschiede zwischen den QuickBASIC-Versionen 2.0 und 4.5. Solange nicht anders erwähnt, beziehen sich Unterschiede zwischen den Versionen 2.0 und 4.5 auch auf Unterschiede zwischen den Versionen 3.0 und 4.5. Die Unterschiede zwischen den Versionen 4.0 und 4.5 bestehen hauptsächlich in der QuickBASIC-Umgebung.

Dieser Anhang enthält folgende Informationen zu Verbesserungen von QuickBASIC Version 4.5:

- Produktfähigkeiten und -eigenschaften
- Erweiterungen der Umgebung
- Verbesserungen beim Kompilieren und Debuggen
- Sprachänderungen
- Datei-Kompatibilität

B.1 Eigenschaften von QuickBASIC

Dieser Abschnitt vergleicht die Eigenschaften der Microsoft QuickBASIC-Version 4.5 mit denen der vorhergehenden Versionen. Die Eigenschaften werden in Tabelle B.1 aufgeführt und nachstehend beschrieben.

Tabelle B.1 Eigenschaften der Microsoft QuickBASIC-Version 4.5

	QuickBASIC-Version			
Eigenschaft	***2.0***	***3.0***	***4.0***	***4.5***
Angeschlossener QB-Ratgeber (ausführliche Referenz)	Nein	Nein	Nein	Ja
Wählbare rechte Maustastenfunktion	Nein	Nein	Nein	Ja
Befehl **Aktuellen Wert anzeigen** für Variablen und Ausdrücke	Nein	Nein	Nein	Ja
Standardsuchpfade festlegen	Nein	Nein	Nein	Ja
Benutzerdefinierte Typen	Nein	Nein	Ja	Ja
IEEE-Format, Unterstützung des mathematischen Koprozessors	Nein	Ja	Ja	Ja
Direkte Hilfe	Nein	Nein	Ja	Ja
Hilfe mit den Symbolen	Nein	Nein	Nein	Ja
Lange (32-Bit) Ganzzahlen	Nein	Nein	Ja	Ja
Zeichenketten fester Länge	Nein	Nein	Ja	Ja
Syntaxüberprüfung bei der Eingabe	Nein	Nein	Ja	Ja
Binär-Datei-E/A	Nein	Nein	Ja	Ja
FUNCTION-Prozeduren	Nein	Nein	Ja	Ja
CodeView-Unterstützung	Nein	Nein	Ja	Ja
Kompatibilität mit anderen Sprachen	Nein	Nein	Ja	Ja
Mehrere Module im Speicher	Nein	Nein	Ja	Ja

Eigenschaft	***QuickBASIC-Version*** 2.0	3.0	4.0	4.5
Kompatibilität mit ProKey SideKick,und SuperKey	Nein	Ja	Ja	Ja
Einfüge-/Überschreibemodus	Nein	Ja	Ja	Ja
Tastaturschnittstelle im WordStar-Stil	Nein	Nein	Ja	Ja
Rekursion	Nein	Nein	Ja	Ja
Fehler-Listings während separater Kompilierung	Nein	Ja	Ja	Ja
Assembler-Listings während separater Kompilierung	Nein	Ja	Ja	Ja

B.1.1 Neue Eigenschaften von QuickBASIC 4.5

Sie haben jetzt Zugriff auf angeschlossene Hilfsdateien für weitere Informationen über QuickBASIC Schlüsselwörter, Befehle und Menüs, sowie allgemeine Themen oder eigene Variablen. Die in den Hilfsdateien vorhandenen Beispiele können kopiert und direkt im eigenen Programm eingefügt werden, um die Entwicklungszeit zu reduzieren.

Mausbenutzer haben die Möglichkeit, die Funktion der rechten Maustaste mit Hilfe des Befehls **Rechte Maustaste** aus dem Menü **Optionen** einzustellen. Es ist die den eigenen Bedürfnissen am besten angepaßte Funktion zu wählen. Weitere Informationen entnehmen Sie bitte Kapitel 19, "Das Menü Aufrufe" im Handbuch *Lernen und Anwenden von Microsoft QuickBASIC.*

Zur Beschleunigung der Fehlerbeseitigung bietet QuickBASIC jetzt den Befehl **Aktuellen Wert anzeigen**, um den Wert einer Variablen oder die Bedingung (wahr oder falsch) eines Ausdruckes sofort erkennen zu können. Nähere Informationen finden Sie in Kapitel 18, "Das Menü Debug" des Handbuches *Lernen und Anwenden von Microsoft QuickBASIC.*

Die Version 4.5 ermöglicht auch die Festlegung von Standardsuchpfaden zu bestimmten Dateitypen. Diese Eigenschaft dient der typenmäßigen Organisation von Dateien und deren Speicherung in getrennten Verzeichnissen. Nach Festlegung des neuen Standardsuchpfades sucht QuickBASIC nach dem richtigen Verzeichnis. Standardpfade können für ausführbare, Include-, Bibliothek- und Hilfsdateien festgelegt werden.

B.1.2 In QuickBASIC 4.0 eingeführte Eigenschaften

Falls Ihnen QuickBASIC nicht bekannt oder Version 4.0 nicht geläufig ist, wäre es angebracht, folgende in QuickBASIC 4.0 eingeführte und von der aktuellen Version unterstützte Eigenschaften durchzusehen.

B.1.2.1 Benutzerdefinierte Typen

Die Anweisung **TYPE** ermöglicht die Erstellung zusammengesetzter Datentypen aus einfachen Datenelementen. Solche Datentypen sind den Strukturen der Sprache C ähnlich. Benutzerdefinierte Typen werden in Kapitel 3, "Datei- und Geräte-E/A" eingehend erläutert.

B.1.2.2 IEEE-Format und Unterstützung des mathematischen Koprozessors

Microsoft QuickBASIC bietet Zahlen im IEEE-Format sowie Unterstützung eines mathematischen Koprozessors an. Wenn kein Koprozessor vorhanden ist, emuliert QuickBASIC den Koprozessor.

Von Programmen ausgeführte Berechnungen, die mit Hilfe der QuickBASIC Version 4.0 kompiliert wurden, sind im allgemeinen genauer und können möglicherweise andere Ergebnisse hervorbringen, als die von Programmen, die unter BASICA oder früheren QuickBASIC-Versionen laufen. Zahlen einfacher Genauigkeit im IEEE-Format bieten größere Genauigkeit durch eine zusätzliche Dezimalstelle. Im Vergleich zu Zahlen doppelter Genauigkeit im Microsoft-Binär-Format bieten Zahlen doppelter Genauigkeit im IEEE-Format ein oder zwei zusätzliche Ziffern für die Mantisse und vergrößern den Bereich des Exponenten.

Es gibt zwei Möglichkeiten, QuickBASIC Version 4.0 und 4.5 mit alten Daten und Programmen zu verwenden:

1. Verwenden Sie die Option /MBF. Auf diese Art und Weise können Sie alte Programme und Datendateien verwenden, ohne die Programme umzuschreiben oder die Dateien zu ändern.
2. Verändern Sie die Datendateien und kompilieren Sie die Programme mit Hilfe der neuen QuickBASIC Version neu. Auf lange Sicht gewährleistet dieses Verfahren die Kompatibilität mit zukünftigen QuickBASIC-Versionen und die Erstellung schnellerer Programme. Es müssen nur Direktzugriffsdateien geändert werden, die reelle Zahlen im Binär-Format enthalten. Dateien, die nur Ganzzahlen oder Zeichenkettendaten enthalten, können unverändert verwendet werden. Nachstehend finden Sie weitere Informationen zu diesen Optionen.

Hinweis Falls aus dem jeweiligen Programm Assembler-Prozeduren aufgerufen werden, die reelle Zahlen verwenden, müssen die Prozeduren so geschrieben sein, daß sie Zahlen im IEEE-Format verwenden. Diese Schreibweise stellt den Standard für den Microsoft Macro Assembler (MASM) Version 5.0 und spätere Versionen dar. Mit früheren Versionen ist die Befehlszeilen-Option /R oder die Assembler-Direktive 8087 zu verwenden.

B.1.2.3 Bereiche der IEEE-Formatzahlen

IEEE-Formatzahlen umfassen einen größeren Bereich als Zahlen im Microsoft-Binär-Format, wie aus folgender Liste ersichtlich:

Zahlentyp	*Wertebereich*
Einfache Genauigkeit	$-3{,}37 * 10^{38}$ bis $-8{,}43 * 10^{-37}$ Wirklich Null $8{,}43 * 10^{-37}$ bis $3{,}37 * 10^{38}$
Doppelte Genauigkeit	$-1{,}67 * 10^{308}$ bis $-4{,}19 * 10^{-307}$ Wirklich Null $4{,}19 * 10^{-307}$ bis $1{,}67 * 10^{308}$

Werte einfacher Genauigkeit sind auf nahezu sieben Ziffern genau. Werte doppelter Genauigkeit sind entweder auf 15 oder auf 16 Ziffern genau.

Beachten Sie, daß Werte doppelter Genauigkeit drei Ziffern im Exponenten haben können. Dies könnte in **PRINT USING**-Anweisungen Probleme verursachen.

B.1.2.4 PRINT USING und Zahlen im IEEE-Format

Da Zahlen doppelter Genauigkeit im IEEE-Format größere Exponenten haben können als Zahlen doppelter Genauigkeit im Microsoft-Binär-Format, kann es sein, daß Sie in **PRINT USING**-Anweisungen ein besonderes Exponentialformat einsetzen müssen. Falls das jeweilige Programm Werte mit drei Ziffern im Exponenten ausdruckt, ist das neue Format zu benutzen. Zum Ausdrucken von Zahlen mit drei Ziffern im Exponenten, müssen fünf anstelle von vier Einfügezeichen (^^^^^) verwendet werden, um das Exponentialformat zu kennzeichnen:

```
PRINT USING "+#.######^^^^^", Umfang#
```

Falls ein Exponent zu groß für ein Feld ist, ersetzt QuickBASIC die erste Ziffer mit einem Prozentzeichen (%), um das Problem anzuzeigen.

B.1.3 Neukompilieren alter Programme mit /MBF

Alte Programme und Dateien arbeiten mit der QuickBASIC Version 4.5 ohne Änderung, wenn die Programme mit der Befehlszeilen-Option /MBF neu kompiliert werden. Um zum Beispiel das als *umwandlg.bas* bezeichnete Programm zu kompilieren, geben Sie folgendes nach der DOS-Eingabeaufforderung ein:

```
BC umwandlg /MBF;
```

Binden Sie das Programm anschließend wie gewohnt. Um mit dem Befehl **EXE-Datei erstellen** neu zu kompilieren, ist die Option /MBF beim Starten von QuickBASIC zu verwenden. Anschließend können Sie wie gewohnt kompilieren.

Die Option /MBF konvertiert Zahlen im Microsoft-Binär-Format in das IEEE-Format beim Lesen aus einer Direktzugriffsdatei und konvertiert sie dann vor Schreiben in eine Datei in das Microsoft-Binär-Format zurück. Diese Operation ermöglicht die Durchführung von Berechnungen mit IEEE-Formatzahlen, während die Zahlen in den Dateien im Microsoft-Binär-Format bleiben.

B.1.4 Konversion von Dateien und Programmen

Wenn Sie Programme und Datendateien ohne den Einsatz der Befehlszeilen-Option /MBF konvertieren möchten, verfahren Sie wie folgt:

1. Kompilieren Sie die Programme neu.
2. Konvertieren Sie die Datendateien.

Die alten QuickBASIC-Programme lassen sich ohne Änderungen kompilieren. Verwenden Sie jedoch nicht die Option /MBF beim Neukompilieren. Das Programm arbeitet nicht mit den neuen Datendateien, falls Sie die Option /MBF verwenden.

Datendateien, die reelle Zahlen enthalten, müssen so konvertiert werden, daß die Zahlen im IEEE-Format gespeichert werden. Die QuickBASIC Version 4.5 enthält acht Funktionen, die Ihnen dabei helfen.

Hinweis Sie brauchen die Datendateien nicht zu konvertieren, wenn diese nur Ganzzahlen und Zeichenkettendaten enthalten. Nur Dateien, die reelle Zahlen enthalten, müssen konvertiert werden.

Version 4.5 stellt für das Lesen bzw. Schreiben reeller Zahlen aus bzw. auf Direktzugriffsdateien die bekannten Funktionen **CVS**, **CVD**, **MKS$** und **MKD$** zur Verfügung. Diese Funktionen behandeln jedoch reelle in den Dateien gespeicherte Zahlen, als IEEE-Formatzahlen, nicht als Microsoft-Binär-Formatzahlen. Zur Behandlung von Zahlen im Microsoft-Binär-Format bietet Version 4.5 die Funktionen **CVSMBF**, **CVDMBF**, **MKSMBF$** und **MKDMBF$** an.

Mit Hilfe dieser Funktionen können Sie ein kurzes Programm schreiben, das Datensätze aus der alten Datei liest (und, falls notwendig, das Microsoft-Binär-Format verwendet), die reellen Zahlen in das IEEE-Format konvertiert, diese Zahlen in einem neuen Datensatz ablegt und den neuen Datensatz ausgibt.

Beispiele

Das folgende Programm kopiert eine alte Datendatei in eine neue Datei und führt die notwendigen Konvertierungen durch:

```
' Definiere Typen für alten und neuen Dateipuffer:
TYPE AltPuffer
   MessReihe AS STRING*20
   XPos AS STRING*4
   YPos AS STRING*4
   FunkWert AS STRING*8
END TYPE

TYPE NeuPuffer
   MessReihe AS STRING*20
   XPos AS SINGLE
   YPos AS SINGLE
   FunkWert AS DOUBLE
END TYPE

' Deklariere Puffer:
DIM AltDatei AS AltPuffer, NeuDatei AS NeuPuffer

' Öffne alte und neue Datendatei:
OPEN "altmbf.dat" FOR RANDOM AS #1 LEN=LEN(AltDatei)
OPEN "neuieee.dat" FOR RANDOM AS #2 LEN=LEN(NeuDatei)

I=1

' Lies den ersten alten Satz:
GET #1,I,AltDatei

DO UNTIL EOF(1)

' Bewege die Feldinhalte in die neuen Datensatzfelder,
' konvertiere dabei die Inhalte von Feldern reeller Zahlen:
   NeuDatei.MessReihe=AltDatei.MessReihe
   NeuDatei.XPos=CVSMBF(AltDatei.XPos)
   NeuDatei.YPos=CVSMBF(AltDatei.YPos)
   NeuDatei.FunkWert=CVDMBF(AltDatei.FunkWert)

' Schreib die konvertierten Felder in die neue Datei:
   PUT #2,I,NeuDatei
   I=I+1
```

```
' Lies den nächsten Datensatz aus der alten Datei:
   GET #1,I,AltDatei
LOOP
CLOSE #1, #2
```

Jeder Datensatz der beiden Dateien hat vier Felder: ein Kennzeichenfeld, zwei Felder, die reelle Zahlen einfacher Genauigkeit enthalten sowie ein letztes Feld, das eine reelle Zahl doppelter Genauigkeit enthält.

Die meiste Konvertierungsarbeit wird mit den Funktionen **CVDMBF** und **CVSMBF** erledigt. Die folgende Programmzeile konvertiert zum Beispiel das Feld doppelter Genauigkeit:

```
NeuDatei.FunkWert=CVDMBF(AltDatei.FunkWert)
```

Die acht Bytes des Datensatzfeldes `AltDatei.FunkWert` werden mit der Funktion **CVDMBF** von einem Wert doppelter Genauigkeit im Microsoft-Binär-Format in einen Wert doppelter Genauigkeit im IEEE-Format umgewandelt. Dieser Wert wird in dem entsprechenden Feld des neuen Datensatzes abgelegt, der später von der Anweisung **PUT** in die neue Datei geschrieben wird.

B.1.5 Andere QuickBASIC Eigenschaften

QuickBASIC 4.0 hat nachstehende Eigenschaften und Werkzeuge eingeführt, die QuickBasic zu einem Entwicklungspaket gemacht haben, das von Profis geschätzt wird, ohne dabei den Anfänger zu überfordern. QuickBASIC 4.5 unterstützt alle diese Eigenschaften, welche inzwischen noch verfeinert wurden.

B.1.5.1 Lange (32-Bit-) Ganzzahlen

Lange (32-Bit) Ganzzahlen beseitigen Rundungsfehler in Berechnungen mit Zahlen, die in dem Bereich -2.147.483.648 bis 2.147.483.647 liegen und ermöglichen erheblich schnellere ganzzahlige Berechnungen als Gleitkommazahlen in diesem Bereich.

B.1.5.2 Zeichenketten fester Länge

Zeichenketten fester Länge ermöglichen die Eingliederung von Zeichenkettendaten in benutzerdefinierten Typen. Weitere Informationen finden Sie in Kapitel 4, "Zeichenkettenverarbeitung".

B.1.5.3 Syntaxüberprüfung bei der Eingabe

Wenn die Syntaxüberprüfung eingeschaltet ist, überprüft QuickBASIC bei der Eingabe jede Zeile auf Syntax- und doppelte Definitions-Fehler. Eine Erläuterung der Syntaxüberprüfung und anderer Eigenschaften des "intelligenten" Editors finden Sie in Kapitel 12, "Die Anwendung des Editors" des Handbuches *Lernen und Anwenden von Microsoft QuickBASIC*.

B.1.5.4 Binär-Datei-E/A

Die Versionen 4.0 und 4.5 erlauben binären Zugriff auf Dateien. Diese Eigenschaft ist beim Lesen und Verändern von Dateien behilflich, die in einem Nicht-ASCII-Format abgespeichert sind. Der Hauptvorteil des Binärzugriffs besteht in der Tatsache, daß die Datei nicht als Ansammlung von Datensätzen behandelt werden muß. In einer im Binärmodus geöffneten Datei können Sie sich auf jede Byte-Position vor- und rückwärts in der Datei bewegen und anschließend eine beliebige Anzahl von Bytes lesen oder schreiben. Zum Unterschied vom Direktzugriff, bei dem die Byteanzahl durch die vordefinierte Länge eines einzelnen Datensatzes festgelegt ist, können verschiedene E/A-Operationen in derselben Datei eine unterschiedliche Anzahl von Bytes lesen bzw. schreiben.

Weitere Informationen über den Zugriff auf Binär-Dateien finden Sie in Kapitel 3, "Datei- und Geräte-E/A".

B.1.5.5 FUNCTION-Prozeduren

FUNCTION-Prozeduren ermöglichen die Ablage einer Funktion in einem Modul und deren Aufruf aus einem anderen Modul. Weitere Informationen zur Verwendung von **FUNCTION**-Prozeduren finden Sie in Kapitel 2, "Prozeduren: Unterprogramme und Funktionen".

In den Versionen vor 4.0 konnten eine **SUB**-Prozedur und eine Variable denselben Namen haben. Jetzt müssen **SUB**- und **FUNCTION**-Prozeduren eindeutig benannt werden.

B.1.5.6 Unterstützung für den CodeView®-Debugger

Sie können die Befehlszeilen-Kommandos *bc.exe* und *link.exe* zur Erstellung ausführbarer Dateien einsetzen, die zu dem Microsoft CodeView-Debugger, kompatibel sind. Es handelt sich um ein leistungsfähiges Werkzeug, das mit dem Microsoft Macro Assembler (Version 5.0 oder spätere Version) bzw. mit Microsoft C (Version 5.0 oder spätere Version) geliefert wird. Die mit BC kompilierten Module können dabei mit Modulen, die mit anderen Microsoft-Sprachen kompiliert sind, gebunden werden, so daß die endgültige ausführbare Datei mit dem CodeView-Debugger untersucht werden kann. Weitere Informationen finden Sie im Anhang G, "Kompilieren und Binden aus DOS".

B.1.5.7 Kompatibilität zu anderen Sprachen

QuickBASIC Version 4.5 ermöglicht den Aufruf von Routinen, die in anderen Microsoft-Sprachen geschrieben sind, anhand der C- oder Pascal-Aufrufvereinbarungen. Argumente werden als kurze (NEAR) bzw. lange (FAR) Referenz oder als Wert übergeben. Anderssprachiger Code kann in einer Quick-Bibliothek abgelegt oder in ausführbaren Dateien gebunden werden.

Die Routine PTR86 wird von QuickBASIC 4.5 nicht unterstützt; an ihrer Stelle sind die Funktionen **VARPTR** und **VARSEG** zu verwenden.

B.1.5.8 Mehrere Module im Speicher

Es können gleichzeitig mehrere Programmodule in den Speicher geladen werden. Die der Version 4.0 vorangehenden Versionen ließen nur ein Modul im Speicher zu. Nun können mehrmodulige Programme innerhalb der QuickBASIC-Umgebung bearbeitet, ausgeführt und debuggt werden. Weitere Informationen zur Verwendung von Mehrfachmodulen entnehmen Sie bitte Kapitel 2, "Prozeduren: Unterprogramme und Funktionen" und Kapitel 7, "Programmieren mit Modulen".

B.1.5.9 Kompatibilität mit ProKey, SideKick und SuperKey

ProKey, SideKick und SuperKey können innerhalb der QuickBASIC-Umgebung eingesetzt werden. Andere die Tastatur umbelegende Software sowie Desktop-Software wird wahrscheinlich nicht mit QuickBASIC zusammen laufen. Informieren Sie sich bei den Lieferanten oder Herstellern anderer Produkte bezüglich der Kompatibilität des jeweiligen Produkts zu QuickBASIC Version 4.5.

B.1.5.10 Einfüge-/Überschreibemodus

Das Betätigen der EINFG-TASTE schaltet den Editiermodus zwischen Einfügen und Überschreiben um. Wenn der Überschreibemodus eingeschaltet ist, wechselt der Cursor von einem blinkenden Unterstreichungszeichen zu einem Rechteck. Beachten Sie, daß EINFG den STRG+O Einfüge-/Überschreibeschalter aus Version 3.0 ersetzt.

Im Einfügemodus fügt der Editor ein eingetipptes Zeichen an der Cursorposition ein. Im Überschreibemodus ersetzt der Editor das Zeichen unter dem Cursor mit dem eingetippten Zeichen. Der Einfügemodus ist der Standardmodus.

B.1.5.11 Tastaturbefehle im WordStar-Stil

QuickBASIC unterstützt viele Tastenfolgen, mit denen Benutzer von WordStar vertraut sind. Eine komplette Liste der Tastaturbefehle im WordStar-Stil finden Sie in Kapitel 12, "Die Verwendung des Editors", des Handbuches *Lernen und Anwenden von Microsoft QuickBASIC*.

B.1.5.12 Rekursion

QuickBASIC Version 4.0 und 4.5 unterstützen die Rekursion, das heißt die Fähigkeit einer Prozedur sich selbst aufzurufen. Rekursion ist bei der Lösung bestimmter Probleme, wie zum Beispiel Sortieren, behilflich. Weitere Informationen zu der Verwendung von Rekursion finden Sie in Kapitel 2, "Prozeduren: Unterprogramme und Funktionen".

B.1.5.13 Fehler-Listings während der getrennten Kompilierung

QuickBASIC zeigt beim Kompilieren der Programme anhand des Befehls BC erläuternde Fehlermeldungen an. Mit Hilfe von BC können Fehlermeldungen in eine Datei oder zu einem Gerät umgelenkt werden, um eine Kopie der Kompilierfehler zu erhalten. Weitere Informationen zu dem Befehl BC finden Sie in Anhang G, "Kompilieren und Binden aus DOS".

Beispiele

Die folgenden Beispiele zeigen, wie die Befehlszeile BC zur Anzeige von Fehlern benutzt wird:

Befehlszeile	***Funktion***
`bc test.bas`	Kompiliert die Datei mit dem Namen *test.bas* und zeigt Fehler auf dem Bildschirm an
`bc test.bas; >test.err`	Kompiliert die Datei *test.bas* und lenkt die Fehlermeldungen in die Datei mit dem Namen *test.err* um.

B.1.5.14 Assembler-Listings während der getrennten Kompilierung

Die Option /A des Befehls BC erstellt ein Listing des von dem Compiler erzeugten Assembler-Codes. Weitere Informationen zu dieser Option entnehmen Sie bitte Anhang G, "Kompilieren und Binden aus DOS".

B.2 Unterschiede in der Umgebung

Die QuickBASIC Programmierumgebung bietet jetzt flexiblere Auswahlmöglichkeiten für Befehle, zusätzliche Fenster-Optionen sowie mehr Menübefehle. Die Abschnitte B.2.1 bis B.2.5 beschreiben die Unterschiede in der Programmierumgebung zwischen Version 4.5 und früheren Versionen.

B.2.1 Befehls- und Optionsauswahl

QuickBASIC Version 4.5 bietet Flexibilität bei der Auswahl von Befehlen aus Menüs bzw. Optionen aus Dialogfeldern.

Version 4.5 erlaubt die Auswahl jedes Menüs durch die Betätigung der ALT-Taste und einer mnemonischen Taste. Jeder Menübefehl und jede Dialogfeld-Option hat ebenfalls jeweils eine mnemonische Taste, die sofort einen Befehl ausführt oder eine Position auswählt. Die mnemonische Taste erscheint auf dem Bildschirm farblich unterstrichen.

In Version 4.5 funktioniert die EINGABETASTE genau wie die LEERTASTE in der Version 3.0 und den früheren Versionen. Sie können einen Befehl aus einem Dialogfeld durch Betätigen der EINGABETASTE ausführen.

B.2.2 Fenster

Version 4.5 läßt bis zu zwei als Arbeitsfenster bezeichnete Arbeitsbereiche, ein Hilfsfenster und ein separates Direkt-Fenster zu. Die der Version 4.0 vorangehenden Versionen unterstützten nur zwei Fenster: den Arbeitsbereich und das Fenster für Fehlermeldungen.

B.2.3 Das neue Menü

QuickBASIC Version 4.5 verfügt über ein neues Menü, mit dem Namen **Optionen**. Das Menü **Optionen** greift auf besondere QuickBASIC Kontrollen zu, die es erlauben, Bildschirmattribute, die Funktion der rechten Maustaste, den Standardpfad zu gegebenen Dateitypen, die Syntaxüberprüfung und die Kontrolle des vollständigen Menüs zu ändern.

B.2.4 Menübefehle

Die bereits in früheren QuickBASIC Versionen vorhandenen Menüs **Debug** und **Hilfe** verfügen über einige neue Befehle. Tabelle B.2 führt sowohl die neuen Befehle dieser Menüs als auch die Befehle des neuen Menüs **Optionen** auf.

Tabelle B.2 Menüs mit neuen Befehlen in der QuickBASIC Version 4.5

Menü	*Befehl*	*Beschreibung*
Debug	**Variable anzeigen**	Fügt dem Debug-Fenster Variablen oder Ausdrücke hinzu.
	Aktuellen Wert anzeigen (neu)	Überprüft sofort den Wert einer Variablen oder eines Ausdruckes.
	Stoppbedingung	Fügt dem Debug-Fenster eine Stoppbedingung hinzu.
	Anzeigevariable löschen	Entfernt selektiv Posten aus dem Debug-Fenster.
	Alle Anzeigevariablen löschen	Entfernt alle Posten des Debug-Fensters.
	Verfolgen ein	Schaltet die Ablaufverfolgung ein und aus.
	Rückverfolgung ein	Zeichnet die 20 zuletzt ausgeführten Anweisungen auf.
	Haltepunkt ein/aus	Schaltet einen Haltepunkt der laufenden Zeile ein und aus.
	Alle Haltepunkte löschen	Entfernt alle Haltepunkte.
	Halt bei Fehler (neu)	Hält die Programmausführung beim Auftreten eines Fehlers unabhängig von der jeweiligen Fehlerbehandlung an.
	Nächste Anweisung festlegen	Legt die Ausführung der nächsten Anweisung bei der Wiederaufnahme eines unterbrochenen Programmablaufes fest.
Optionen (neu)	**Bildschirm (neu)**	Paßt Bildschirmelemente den Erfordernissen des Benutzers an.
	Suchpfade festlegen (neu)	Ändert die Standardsuchpfade der angegebenen Dateitypen.
	Rechte Maustaste (neu)	Wählt die Funktion der rechten Maustaste aus.
	Syntaxüberprüfung	Schaltet die automatische Syntaxüberprüfung ein und aus.
	Vollständiges Menü (neu)	Schaltet zwischen vollständigem Menü und Kurzmenü um.

Fortsetzung auf der folgenden Seite.

Menü	*Befehl*	*Beschreibung*
Hilfe	**Index (neu)**	Zeigt eine alphabetische Liste der QuickBASIC Schlüsselworte und deren Kurzbeschreibung an.
	Themen (neu)	Zeigt eine visuelle Übersicht der Inhalte an.
	Begriff	Erteilt Informationen über Variablen und Schlüsselworte.
	Überblick (neu)	Erläutert die Verwendung des Hilfsmenüs und der üblichen Tastenkombinationen zur direkten Befehlseingabe.

Die Option "**Debug**" der Version 3.0 wurde aus dem Menü Ausführen entfernt. In Version 4.5 können Sie jederzeit mit Hilfe der Debug-Befehle des Menüs **Debug** Fehler entfernen.

B.2.5 Änderungen der Tastenkombinationen beim Bearbeiten

Die Tastaturschnittstelle der QuickBASIC Version 4.5 wurde erweitert und enthält nun Tastenkombinationen, die denen des WordStar-Editors ähnlich sind. (Nähere Informationen über die Bearbeitung sind Kapitel 12, "Die Verwendung des Editors", und Kapitel 13, "Das Menü Bearbeiten", des Handbuches *Lernen und Anwenden von Microsoft QuickBASIC* zu entnehmen.) Die Funktionen, die von den in Tabelle B.3 aufgeführten Tastenfolgen in QuickBASIC-Version 2.0 durchgeführt werden, sind in den QuickBASIC-Versionen 4.0 und 4.5 geändert.

Tabelle B.3 Änderungen der Bearbeitungs-Tasten

Funktion	*Taste in QuickBASIC 2.0*	*Taste in QuickBASIC 4.5*
Rückgängig	UMSCHALTTASTE+ESC	ALT+RÜCKTASTE
Ausschneiden	ENTF	UMSCHALTTASTE+ENTF
Kopieren	F2	STRG+EINFG
Einfügen	EINFG	UMSCHALTTASTE+EINFG
Löschen	---	ENTF
Überschreiben	---	EINFG

B.3 Unterschiede beim Debuggen und Kompilieren

In der QuickBASIC-Programmierumgebung der Version 4.5 sind Kompilieren und Debuggen keine separaten Operationen. Das Programm ist jederzeit ausführbar, und Fehler im Code können auf verschiedene Weise während der Programmierung beseitigt werden. Dieser Abschnitt beschreibt die Unterschiede der Kompilier- und Debug-Eigenschaften zwischen den QuickBASIC Versionen 4.0 und 4.5 und den früheren Versionen.

B.3.1 Unterschiede in der Befehlszeile

Die QuickBASIC Version 4.0 und 4.5 unterstützen Befehlszeilen-Optionen der Version 2.0 nur für den Befehl QB, nicht auch für den Befehl BC. Um ein Programm außerhalb der QuickBASIC-Umgebung zu kompilieren, ist der in Anhang G, "Kompilieren und Binden aus DOS", beschriebene Befehl BC zu verwenden.

Die Versionen 4.0 und 4.5 erfordern keine der Kompilier-Optionen, die in Tabelle 4.1 des Handbuches zu Microsoft QuickBASIC Version 2.0 aufgeführt sind. Beim Versuch QuickBASIC mit Hilfe dieser Optionen als Befehlszeilen-Optionen aufzurufen, erscheint eine Fehlermeldung. Ähnlich ist es in Version 3.0 notwendig, bestimmte Kompilier-Optionen aus dem Dialogfeld "Compile" zu wählen; dieses Verfahren wird nun automatisch erledigt. Tabelle B.4 beschreibt die Art und Weise, in der QuickBASIC jetzt die Funktionen dieser Optionen unterstützt.

Tabelle B.4 In den QuickBASIC Versionen 4.0 oder 4.5 nicht verwendete QB und BC Optionen

Version 2.0	***Version 4.5***
On Error (/E)	Wird automatisch gesetzt, wenn eine **ON ERROR**-Anweisung vorhanden ist.
Debug (/D)	Ist immer beim Starten eines Programms innerhalb der QuickBASIC-Umgebung eingeschaltet. Bei der Erstellung von ausführbaren Programmen auf Diskette oder in Quick-Bibliotheken ist die Option **Debug-Code erstellen** zu benutzen.

Fortsetzung auf der folgenden Seite.

Version 2.0	*Version 4.5*
Überprüfung zwischen Anweisung (/V) und Ereigniserfassung (/W)	Ist automatisch gesetzt, wenn eine Anweisung **ON** *Ereignis* vorhanden ist.
Resume Next (/X)	Ist automatisch gesetzt, wenn eine **RESUME NEXT**-Anweisung vorhanden ist.
Datenfelder in Zeilenanordnung (/R)	Verfügbar nur bei Kompilierung mit BC.
Minimieren von Zeichenkettendaten (/S)	Der Standard für QB. Um diese Option auszuschalten, muß mit BC von der Befehlszeile aus kompiliert werden.

Die in Tabelle B.5 aufgeführten Optionen sind jetzt für die QB- und BC-Befehle verfügbar:

Tabelle B.5 In Version 4.0 eingeführte Optionen für die Befehle QB und BC

Option	*Beschreibung*
/AH	Ermöglicht es dynamischen Feldern, welche Datensätze, Zeichenketten fester Länge, sowie numerischen Daten enthalten, jeweils größer als 64K zu sein. Falls diese Option nicht angegeben ist, beträgt die maximale Größe pro Datenfeld 64K. Beachten Sie, daß sich diese Option auf die Datenübertragung an Prozeduren nicht auswirkt. (Diese Option wird mit den Befehlen QB und BC verwendet.)
/H	Zeigt mit der auf der jeweiligen Hardware höchstmöglichen Auflösung an. Wenn zum Beispiel ein EGA vorhanden ist, zeigt QuickBASIC 43 Textzeilen und 80 Spalten an. (Diese Option wird nur mit dem Befehl QB verwendet.)
/MBF	Konvertiert Zahlen im Microsoft-Binär-Format in das IEEE-Format. Weitere Informationen zu dieser Option sind in Abschnitt B.1.2.3 zu finden. (Diese Option wird mit den Befehlen QB und BC verwendet.)
/RUN *Quelldatei*	Startet *Quelldatei* sofort, ohne zunächst die QuickBASIC-Programmierumgebung anzuzeigen. (Diese Option wird nur mit dem Befehl QB verwendet.)

B.3.2 Unterschiede bei separater Kompilierung

Die Versionen 4.0 und 4.5 lassen keine separate Kompilierung mit dem Befehl QB zu. Verwenden Sie den in Anhang G, "Kompilieren und Binden aus DOS", beschriebenen Befehl BC, um Dateien zu kompilieren und zu binden, ohne die Programmierumgebung aufzurufen.

B.3.3 Benutzerbibliotheken und BUILDLIB

Für frühere Versionen erstellte Benutzerbibliotheken sind nicht kompatibel zu den Versionen 4.0 und 4.5. Die Bibliothek muß aus den ursprünglichen Quelldateien neu aufgebaut werden.

Benutzerbibliotheken werden jetzt als Quick-Bibliotheken bezeichnet. Es gibt keinen Unterschied in ihrer Funktion oder Verwendung. Die Erweiterung des Dateinamens dieser Bibliotheken ist nun *.qlb* anstelle von *.exe* Das Hilfsprogramm BUILDLIB wird nicht länger benötigt. Quick-Bibliotheken werden nun aus der Programmierumgebung oder von der link-Befehlszeile angelegt. Weitere Informationen finden Sie in Anhang H, "Erstellung und Verwendung der Quick-Bibliotheken".

B.3.4 Einschränkungen für Include-Dateien

Include-Dateien dürfen nur **SUB**- oder **FUNCTION**-Prozedurvereinbarungen, aber keine Definitionen enthalten. Wenn Sie eine alte Include-Datei mit Prozedurdefinitionen verwenden müssen, fügen Sie die Include-Datei mit Hilfe des Befehls **Zusammenführen** aus dem Dateimenü in das aktuelle Modul ein. Wenn Sie eine Include-Datei einfügen, die eine **SUB**-Prozedur enthält, erscheint der Text der Prozedur nicht im momentan aktiven Fenster. Um ihren Text anzuzeigen oder zu bearbeiten, wählen Sie das Kommando **SUBs** aus dem Menü **Ansicht** und danach den Prozedurnamen aus dem Verzeichnisfeld. Nachdem der Text zusammengeführt ist, kann das Programm gestartet werden.

Andererseits kann es sein, daß Sie sich zur Ablage der **SUB**-Prozedur in einem separaten Modul entschließen. In diesem Fall müssen Sie einen der folgenden zwei Schritte für jede gemeinsam benutzte Variable (Variablen, die in einer **COMMON SHARED**- oder **[RE]DIM SHARED**-Anweisung außerhalb der **SUB**-Prozedur oder in einer **SHARED**-Anweisung innerhalb der **SUB**-Prozedur deklariert sind) durchführen, da auf diese Weise deklarierte Variablen nur innerhalb eines einzigen Moduls gemeinsam benutzt werden:

1. Benutzen Sie die Variablen zwischen den Modulen gemeinsam, indem Sie die Variablen in beiden Modulen auf Modul-Ebene in **COMMON**-Anweisungen aufführen.
2. Übergeben Sie die Variablen an die **SUB**-Prozedur in einer Argumentenliste.

Eine ausführliche Erklärung zu Include-Dateien und Modulen in QuickBASIC finden Sie in Kapitel 2, "Prozeduren: Unterprogramme und Funktionen", und in Kapitel 7, "Programmieren mit Modulen".

B.3.5 Debuggen

QuickBASIC unterstützt die schnellere Fehlerbeseitigung der Programme durch folgende Debug-Eigenschaften:

1. Mehrere Haltepunkte.
2. Anzeigeausdrücke, Stoppbedingungen und den Befehl **Aktuellen Wert anzeigen**.
3. Verbesserte Programmverfolgung.
4. Ein Bildschirmfenster, das während des Einzelschrittmodus den Programmtext anzeigt.
5. Die Fähigkeit, Variablenwerte während der Ausführung zu ändern, und die Ausführung anschließend fortzusetzen.
6. Die Fähigkeit, Programme zu bearbeiten und anschließend ohne Neustart mit der Ausführung fortzufahren.
7. Rückverfolgung.
8. Das Menü **Aufrufe**.
9. Eine Symbolhilfe.

Die in Tabelle B.6 aufgeführten Funktionstastenkombinationen zur Fehlersuche haben sich in Vergleich zur QuickBASIC-Version 2.0 geändert:

Tabelle B.6 Änderungen der Debug-Tasten

Funktion	***Taste in der QuickBASIC Version 2.0***	***Taste in der QuickBASIC Version 4.5***
Verfolgen	ALT+F8	F8
Einzelschritt	ALT+F9	F10

Beachten Sie, daß der Animationsmodus beim Umschalten des Befehls **Verfolgen ein** aus dem Menü **Debug** eingeschaltet wird, starten Sie danach Ihr Programm.

Weitere Informationen finden Sie in Kapitel 9, "Debuggen während der Programmierung", des Handbuches *Lernen und Anwenden von Microsoft QuickBASIC*.

B.4 Änderungen der Sprache BASIC

Dieser Abschnitt beschreibt die in QuickBASIC Version 4.5 und früheren Versionen vorgenommenen Erweiterungen und Änderungen der Sprache BASIC. Tabelle B.7 führt die von diesen Änderungen betroffenen Schlüsselwörter und Quick BASIC Versionen auf. Ausführlichere Erläuterungen dieser Änderungen finden Sie im Anschluß an diese Tabelle.

Tabelle B.7 Änderungen der Sprache BASIC

	QuickBASIC Version			
Schlüsselwort	***2.0***	***3.0***	***4.0***	***4.5***
AS	Nein	Nein	Ja	Ja
CALL	Nein	Nein	Ja	Ja
CASE	Nein	Ja	Ja	Ja
CLEAR	Nein	Nein	Ja	Ja
CLNG	Nein	Nein	Ja	Ja
CLS	Nein	Ja	Ja	Ja
COLOR	Nein	Nein	Ja	Ja
CONST	Nein	Ja	Ja	Ja
CVL	Nein	Nein	Ja	Ja
CVSMBF, CVDMBF	Nein	Ja	Ja	Ja
DECLARE	Nein	Nein	Ja	Ja
DEFLNG	Nein	Nein	Ja	Ja
DIM	Nein	Nein	Ja	Ja
DO...LOOP	Nein	Ja	Ja	Ja
EXIT	Nein	Ja	Ja	Ja
FILEATTR	Nein	Nein	Ja	Ja
FREEFILE	Nein	Nein	Ja	Ja
FUNCTION	Nein	Nein	Ja	Ja
GET	Nein	Nein	Ja	Ja

Fortsetzung auf der folgenden Seite.

	QuickBASIC Version			
Schlüsselwort	***2.0***	***3.0***	***4.0***	***4.5***
LCASE$	Nein	Nein	Ja	Ja
LEN	Nein	Nein	Ja	Ja
LSET	Nein	Nein	Ja	Ja
LTRIM$	Nein	Nein	Ja	Ja
MKL$	Nein	Nein	Ja	Ja
MKSMBF$, MKDMBF$	Nein	Ja	Ja	Ja
OPEN	Nein	Nein	Ja	Ja
ON UEVENT	Nein	Nein	Nein	Ja
PALETTE	Nein	Nein	Ja	Ja
PUT	Nein	Nein	Ja	Ja
RTRIM$	Nein	Nein	Ja	Ja
SCREEN	Nein	Nein	Ja	Ja
SEEK	Nein	Nein	Ja	Ja
SELECT CASE	Nein	Ja	Ja	Ja
SETMEM	Nein	Nein	Ja	Ja
SLEEP	Nein	Nein	Nein	Ja
STATIC	Nein	Nein	Ja	Ja
TYPE	Nein	Nein	Ja	Ja
UCASE$	Nein	Nein	Ja	Ja
VARPTR	Nein	Nein	Ja	Ja
VARSEG	Nein	Nein	Ja	Ja
WIDTH	Nein	Ja	Ja	Ja

Der folgende Abschnitt erläutert ausführlicher die Unterschiede der oben zusammengefaßten Schlüsselwörter.

Schlüsselwörter	*Erklärung*
AS	Die Klausel **AS** ermöglicht die Verwendung benutzerdefinierter Typen in **DIM**-, **COMMON**- und **SHARED**-Anweisungen, sowie in **DECLARE**-, **SUB**- und **FUNCTION**-Parameterlisten.
CALL	Die Verwendung von **CALL** ist optional für den Aufruf von Unterprogrammen, die mit der Anweisung **DECLARE** deklariert sind.
CLEAR	Die Anweisung **CLEAR** setzt nicht mehr die Gesamtgröße des Stapels, sondern nur die vom Programm erforderte Stapelgröße fest. QuickBASIC setzt die Stapelgröße auf den von der **CLEAR**-Anweisung angegebenen Betrag, plus den von QuickBASIC selbst benötigten Betrag.
CLNG	Die Funktion **CLNG** rundet ihr eigenes Argument und gibt eine lange (4-Byte) Ganzzahl zurück, die dem Argument entspricht.
CLS	Die Anweisung **CLS** wurde zur Gewährleistung einer größeren Beweglichkeit beim Löschen des Bildschirms verändert. Es ist zu beachten, daß QuickBASIC den Bildschirm am Anfang jedes Programms nicht mehr automatisch löscht, wie das in früheren Versionen der Fall war.
COLOR, SCREEN, PALETTE WIDTH	Die Anweisungen **COLOR**, **SCREEN**, **PALETTE**, **WIDTH** wurden erweitert, um die mit den IBM PS/2 VGA und Multicolor Graphics Array (MCGA) verfügbaren Bildschirmmodi einzuschalten.
CONST	**CONST**-Anweisungen ermöglichen die Definition symbolischer Konstanten zur Verbesserung der Programmlesbarkeit und -pflege.
CVL	Die Funktion **CVL** wird zum Lesen von langen Ganzzahlen verwendet, die in Direktzugriffs-Datendateien als Zeichenketten gespeichert sind. **CVL** konvertiert eine mit der Funktion **MKL$** erzeugte Vier-Byte-Zeichenkette zur Verwendung im BASIC-Programm zurück in eine lange Ganzzahl.

Schlüsselwörter	*Erklärung*
CVSMBF, CVDMBF	Die Funktionen **CVSMBF** und **CVDMBF** konvertieren Zeichenketten, die Zahlen im Microsoft-Binär-Format enthalten, in IEEE-Formatzahlen. Obwohl QuickBASIC Version 4.5 diese Anweisungen unterstützt, sind sie überflüssig, da das IEEE-Format jetzt den QuickBASIC Standard darstellt.
DECLARE	Die Anweisung **DECLARE** ermöglicht den Aufruf von Prozeduren aus unterschiedlichen Modulen, die Überprüfung der Anzahl und des Typs der übergebenen Argumente sowie den Aufruf von Prozeduren vor deren Definierung.
DEFLNG	Die Anweisung **DEFLNG** deklariert alle Variablen, **DEF FN**-Funktionen und **FUNCTION**-Prozeduren als lange Ganzzahlen. Das heißt, daß eine Variable oder Funktion standardmäßig eine lange Ganzzahl ist, wenn sie nicht in einer Klausel als **AS** *Typ* deklariert wurde, oder kein ausdrückliches Typdefinitionssuffix, wie zum Beispiel % oder $ hat.
DIM	Die Klausel **TO** der Anweisung **DIM** ermöglicht die Angabe von Indizes mit beliebigen ganzzahligen Werten, die eine größere Flexibilität bei Datenfelddeklarationen gewährleistet.
DO...LOOP	**DO...LOOP**-Anweisungen stellen leistungsfähigere Schleifen zur Verfügung, die das Schreiben von besser strukturierten Programmen erlauben.
EXIT	**EXIT {DEF | DO | FOR | FUNCTION | SUB}**-Anweisungen bieten bequeme Ausstiegsmöglichkeiten aus Schleifen und Prozeduren.
FREEFILE, FILEATTR	Die Funktionen **FREEFILE** und **FILEATTR** sind beim Schreiben von Anwendungen behilflich, die Datei-E/A in einer mehrmoduligen Umgebung erledigen.
FUNCTION	Die **FUNCTION...END FUNCTION**-Prozedur ermöglicht die Definition einer mehrzeiligen Prozedur, die aus einem Ausdruck aufgerufen werden kann. Diese Prozeduren verhalten sich im wesentlichen wie eingebaute Funktionen, wie zum Beispiel **ABS** oder mehrzeilige **DEF FN**-Funktionen der QuickBASIC-Versionen 1.0 bis 3.0. Zum Unterschied von einer **DEF FN**-Funktion kann eine **FUNCTION**-Prozedur jedoch in einem Modul definiert und aus einem anderen aufgerufen werden. Außerdem kennen **FUNCTION**-Prozeduren lokale Variablen und unterstützen Rekursionen.

Schlüsselwörter	*Erklärung*
GET, PUT	Für E/A-Operationen ist die Syntax der **GET**- bzw. **PUT**-Anweisung erweitert, um mit **TYPE...END TYPE**-Anweisungen definierte Datensätze einzubeziehen, wobei die Verwendung der **FIELD**-Anweisung überflüssig wird.
LCASE$, UCASE$, LTRIM$, RTRIM$	In der Version 4.5 stehen folgende Funktionen zur Verarbeitung von Zeichenketten zur Verfügung:

Funktion	*Rückgabewert*
LCASE$	Liefert eine Kopie der Zeichenkette, wobei alle Buchstaben in Kleinbuchstaben umgewandelt sind.
UCASE$	Liefert eine Kopie der Zeichenkette, wobei alle Buchstaben in Großbuchstaben umgewandelt sind.
LTRIM$	Liefert eine Kopie der Zeichenkette, wobei alle führenden Leerzeichen entfernt sind.
RTRIM$	Liefert eine Kopie der Zeichenkette, wobei alle nachfolgenden Leerzeichen entfernt sind.

LEN	Die Funktion **LEN** wurde erweitert, um die Byteanzahl zurückzugeben, die eine skalare oder Datensatzvariable, eine Konstante, einen Ausdruck oder ein Datenfeldelement benötigt.
LSET	Die Anweisung **LSET** ist erweitert, um sowohl Datensatz- als auch Zeichenkettenvariablen einzubeziehen. Dies ermöglicht es, eine Datensatzvariable einer anderen Datensatzvariablen zuzuweisen, selbst wenn die Datensätze nicht gleich sind.
MKL$	Die Funktion **MKL$** wird zur Umwandlung langer Ganzzahlen in Zeichenketten eingesetzt, die in Direktzugriffs-Datendateien gespeichert werden können. Verwenden Sie die Funktion **CVL**, um die Zeichenkette in eine lange Ganzzahl zurückzuwandeln.
MKSMBF$, MKDMBF$	Die Funktionen **MKSMBF$** und **MKDMBF$** wandeln IEEE-Formatzahlen in Zeichenketten um, die Zahlen im Microsoft-Binär-Format enthalten. Für Version 4.5, die das IEEE-Format benutzt, sind sie überflüssig (werden aber von ihr unterstützt).
ON UEVENT	Die Anweisung **ON UEVENT** lenkt das Programm beim Auftreten eines benutzerdefinierten Ereignisses (**UEVENT**) auf eine gegebene Stelle. Diese Anweisung ist wie alle anderen Anweisungen zur Ereignisbehandlung zu verwenden.

Schlüsselwörter	*Erklärung*

OPEN	Die Anweisung **OPEN** öffnet nun zwei Dateien mit demselben Namen für **OUTPUT** oder **APPEND**, solange sich die Pfadnamen unterscheiden. Zum Beispiel ist nun folgendes zulässig:
	`OPEN "beisp" FOR APPEND AS #1`
	`OPEN "tmp\beisp" FOR APPEND AS #2`
	Der Syntax der **OPEN**-Anweisung ist ein binärer Dateimodus hinzugefügt worden. Informationen zur Verwendung dieses Modus finden Sie in Kapitel 3, "Datei- und Geräte-E/A".

SEEK	Die **SEEK**- Anweisung und -Funktion ermöglichen die Positionierung in einer Datei auf ein beliebiges Byte oder einen beliebigen Datensatz. Weitere Informationen finden Sie in Kapitel 3, "Datei- und Geräte-E/A".
SELECT CASE	**SELECT CASE**-Anweisungen bieten eine Möglichkeit zur Vereinfachung komplexer Abfragebedingungen. Die **CASE**-Klauseln der **SELECT CASE**-Anweisung akzeptieren jetzt jeden Ausdruck (einschließlich variabler Ausdrücke) als Argument; in früheren Versionen waren nur konstante Ausdrücke erlaubt.
SETMEM	Die Funktion **SETMEM** erleichtert mehrsprachige Programmierung, indem sie die Verkleinerung des von BASIC zugewiesenen dynamischen Speicherplatzes erlaubt, so daß dieser von anderssprachigen Prozeduren verwendet werden kann.
SLEEP	Die Anweisung **SLEEP** veranlaßt eine Programmunterbrechung während einer bestimmten Zeitspanne, bis zur nächsten Tastenbetätigung oder bis zum Auftreten eines eingeschalteten Ereignisses. Das optionale Argument gibt die Länge der Unterbrechung in Sekunden an.
STATIC	Das Auslassen des **STATIC**-Attributs aus **SUB**-und **FUNCTION** -Anweisungen bewirkt die Zuweisung von Variablen beim Aufruf, nicht bei der Definition der Prozeduren. Solche Variablen behalten nicht ihre Werte zwischen Prozeduraufrufen.

Schlüsselwörter	*Erklärung*
TYPE	Die Anweisung **TYPE...END TYPE** ermöglicht die Definierung von Datentypen, die Elemente unterschiedlicher einfacher Typen enthalten. Dieser Vorgang vereinfacht die Definierung sowie den Zugriff auf Datensätze in Direktzugriffsdateien, so daß das Auslassen des Schlüsselwortes **STATIC** die Rekursion zuläßt.
UEVENT {PN | STOP |OFF}	Aktiviert, unterbricht oder deaktiviert ein benutzerdefiniertes Ereignis. Die **UEVENT**-Anweisungen werden wie alle anderen Anweisungen der Ereignisverfolgung benutzt.
VARPTR	Die Funktion **VARPTR** gibt nun den ganzzahligen 16-Bit-Offset der BASIC-Variablen oder des BASIC-Datenfeldelementes zurück. Der Offset beginnt am Anfang des Segmentes, das die Variable oder das Datenfeldelement enthält.
VARSEG	Die Funktion **VARSEG** gibt die Segmentadresse ihres eigenen Argumentes zurück. Dies ermöglicht die geeignete Einstellung von **DEF SEG** zur Verwendung mit **PEEK, POKE, BLOAD, BSAVE** sowie **CALL ABSOLUTE.** Außerdem liefert diese Funktion das für die Verwendung mit **CALL INTERRUPT** geeignete Segment bei der Ausführung von Betriebssystem- oder BIOS-Interrupts.
WIDTH	Ein neues Argument der Anweisung **WIDTH** erlaubt dem Programm, die erweiterten Zeilenmodi auf Maschinen zu verwenden, die mit einer EGA-, VGA- oder MCGA-Adapterkarte ausgerüstet sind.

Hinweis Mit der einzeiligen **IF...THEN...ELSE**-Anweisung dürfen **NEXT**- und **WEND**-Anweisungen nicht länger bedingt ausgeführt werden.

B.5 Datei-Kompatibilität

Alle QuickBASIC-Versionen sind Quellcode-kompatibel; ein für eine frühere Version angelegter Quellcode wird von Version 4.5 kompiliert, mit Ausnahme der in Abschnitt B.3.4, "Einschränkungen für Include-Dateien" aufgeführten Punkte. QuickBASIC 4.5 übersetzt die in QuickBASIC 4.0 enthaltenen Binärdateien. Wenn Sie jedoch bei der Übersetzung anderer Binärdateien Schwierigkeiten haben, speichern Sie das Programm im ASCII-Textformat von QuickBASIC 4.0 und laden Sie es mit QuickBASIC 4.5. Für frühere QuickBASIC-Versionen angelegte Objektdateien und Benutzerbibliotheken müssen erneut kompiliert werden.

Anhang C: Einschränkungen für QuickBASIC

QuickBASIC und der BC-Compiler bieten einerseits vielseitige Programmiermöglichkeiten, müssen sich jedoch andererseits auf eine überschaubare Dateigröße und Komplexität beschränken. Aus diesem Grund kann es vorkommen, daß in manchen Situationen die in diesem Anhang aufgeführten Grenzen erreicht werden.

Tabelle C.1 Einschränkungen für QuickBASIC

Namen und Zeichenketten	***Höchstanzahl***	***Mindestanzahl***
Variablennamen	40 Zeichen	1 Zeichen
Zeichenkettenlänge	32.767 Zeichen	0 Zeichen
Ganzzahlen	32.767	-32.768
Lange Ganzzahlen	2.147.483.647	-2.147.483.648
Positive Zahlen einfacher Genauigkeit	3,402823 E+38	1,401298 E-45
Negative Zahlen einfacher Genauigkeit	-1,401298 E-45	-3,402823 E+38
Positive Zahlen doppelter Genauigkeit	1,797693134862315 D+308	4,940656458412465 D-324
Negative Zahlen doppelter Genauigkeit	-4,940656458412465 D-324	-1,797693134862315 D+308

Fortsetzung auf der folgenden Seite.

Datenfelder	***Höchstanzahl***	***Mindestanzahl***
Datenfeldgröße (aller Elemente)		
Statische	65.535 Bytes (64K)	1
Dynamische	Verfügbarer Speicher	
Datenfelddimension	8	1
Datenfeldindizes	32.767	-32.768

Dateien und Prozeduren	***Höchstanzahl***	***Mindestanzahl***
An eine Prozedur übergebene Argumentanzahl	60 interpretierte Argumente	0
Verschachtelung der Include-Dateien	5 Ebenen	0
Größe der interpretierten Prozedur	65.535 Bytes (64K)	0
Größe des kompilierten Moduls	65.535 Bytes (64K)	0
Gleichzeitig geöffnete Datendateien	255	0
Anzahl der Datensätze in einer Datendatei	2.147.483.647	1
Datensatzgröße in Bytes der Datendateien	32.767 Bytes (32K)	1 Byte
Größe der Datendatei	Verfügbarer Diskettenplatz	0
Pfadnamen	127 Zeichen	1 Zeichen
Anzahl der Fehlermeldung	255	1

Editieren in Umgebung der QuickBASIC	***Höchstanzahl***	***Mindestanzahl***
Eintrag im Textfeld	128 Zeichen	0 Zeichen
Zeichenkette "Suchen nach"	128	1
Zeichenkette "Verändern zu"	40	0

Editieren in Umgebung der QuickBASIC	***Höchstanzahl***	***Mindestanzahl***
Punkte zur Textmarkierung	4	0
Stoppbedingungen und/ oder Anzeigeausdrücke	8	0
Zeilenanzahl im Direktfenster	10	0
Zeichenanzahl in einer Zeile des Arbeitsfensters	255	0
Länge der Zeichenketten **COMMAND$**	124	0

Anhang D: Tastaturabfragecodes und ASCII-Zeichencodes

Die folgende Tabelle gibt Ihnen einen Überblick über die in diesem Handbuch verwendeten Tastennamen und die Entsprechungen, die Sie auf einigen Tastaturen an deren Stelle finden.

ALT-TASTE	ALT
RÜCKTASTE	BACKSPACE, BKSP
UNTBR-TASTE	BREAK
UMSCHALT-FESTSTELLTASTE	CAPS LOCK
STRG-TASTE	CONTROL, CTRL
RICHTUNGSTASTEN	CURSOR KEYS
ENTF-TASTE	DELETE, DEL
NACH UNTEN (↓)	DOWN
ENDE-TASTE	END
EINGABETASTE	ENTER
ESC-TASTE	ESCAPE, ESC
POS1-TASTE	HOME
EINFG-TASTE	INSERT, INS
NACH LINKS (←)	LEFT
NUM-FESTSTELLTASTE	NUMLOCK
BILD ↓-TASTE	PAGE DOWN, PG DN
BILD ↑-TASTE	PAGE UP, PG UP
PAUSE-TASTE	PAUSE
ROLLEN-FESTSTELLTASTE	SCROLL LOCK
UMSCHALTTASTE	SHIFT
LEERTASTE	SPACE BAR

S-ABF-TASTE	SYS RQ
DRUCK-TASTE	PRINT SCREEN, PRTSC
NACH RECHTS (→)	RIGHT
TAB-TASTE	TAB
NACH OBEN (↑)	UP

D.1 Tastaturabfragecodes

Die folgende Tabelle listet die DOS Abfragecodes auf, die von der **INKEY$**-Funktion zurückgegeben werden.

Tastenkombinationen, die in der Zeichenspalte NUL enthalten sind, geben 2 Bytes zurück, bzw. ein Nullbyte (&H00) gefolgt von den in den Dez- und Hex-Spalten aufgeführten Werten. Zum Beispiel wird durch die Betätigung der Tastenkombination ALT+F1 ein Nullbyte zurückgegeben, das von einem Byte mit 104 (&H68) gefolgt wird.

Taste	Abfrage-code		ASCII oder Erweitert*			ASCII oder Erweitert* mit UMSCHALT			ASCII oder Erweitert* mit STRG			ASCII oder Erweitert* mit ALT		
	Dez	Hex	Dez	Hex	Zei	Dez	Hex	Zei	Dez	Hex	Zei	Dez	Hex	Zei
ESC	1	01	27	1B		27	1B		27	1B				
1 !	2	02	49	31	1	33	21	!				120	78	NUL
2 @	3	03	50	32	2	64	40	@	3	03	NUL	121	79	NUL
3 #	4	04	51	33	3	35	23	#				122	7A	NUL
4 $	5	05	52	34	4	36	24	$				123	7B	NUL
5 %	6	06	53	35	5	37	25	%				124	7C	NUL
6 ^	7	07	54	36	6	94	5E	^	30	1E		125	7D	NUL
7 &	8	08	55	37	7	38	26	&				126	7E	NUL
8 *	9	09	56	38	8	42	2A	*				127	7F	NUL
9 (	10	0A	57	39	9	40	28	(				128	80	NUL
0)	11	0B	48	30	0	41	29	)				129	81	NUL
- _	12	0C	45	2D	-	95	5F	_	31	1F		130	82	NUL
= +	13	0D	61	3D	=	43	2B	+				131	83	NUL
BKSP	14	0E	8	08		8	08		127	7F				
TAB	15	0F	9	09		15	0F	NUL						
Q	16	10	113	71	q	81	51	Q	17	11		16	10	NUL
W	17	11	119	77	w	87	57	W	23	17		17	11	NUL
E	18	12	101	65	e	69	45	E	5	05		18	12	NUL
R	19	13	114	72	r	82	52	R	18	12		19	13	NUL
T	20	14	116	74	t	84	54	T	20	14		20	14	NUL
Y	21	15	121	79	y	89	59	Y	25	19		21	15	NUL
U	22	16	117	75	u	85	55	U	21	15		22	16	NUL
I	23	17	105	69	i	73	49	I	9	09		23	17	NUL
O	24	18	111	6F	o	79	4F	O	15	0F		24	18	NUL
P	25	19	112	70	p	80	50	P	16	10		25	19	NUL
[{	26	1A	91	5B	[	123	7B	{	27	1B				
] }	27	1B	93	5D	]	125	7D	}	29	1D				
ENTER	28	1C	13	0D	CR	13	0D	CR	10	0A	LF			
CTRL	29	1D												
A	30	1E	97	61	a	65	41	A	1	01		30	1E	NUL
S	31	1F	115	73	s	83	53	S	19	13		31	1F	NUL
D	32	20	100	64	d	68	44	D	4	04		32	20	NUL
F	33	21	102	66	f	70	46	F	6	06		33	21	NUL
G	34	22	103	67	g	71	47	G	7	07		34	22	NUL
H	35	23	104	68	h	72	48	H	8	08		35	23	NUL
J	36	24	106	6A	j	74	4A	J	10	0A		36	24	NUL
K	37	25	107	6B	k	75	4B	K	11	0B		37	25	NUL
L	38	26	108	6C	l	76	4C	L	12	0C		38	26	NUL
; :	39	27	59	3B	;	58	3A	:						
' "	40	28	39	27	'	34	22	"						
` ~	41	29	96	60	`	126	7E	~						

* Erweiterter Code gibt NUL (ASCII 0) als erstes Zeichen zurück. Dieses ist ein Signal, daß ein zweiter (erweiterter) Code im Tastaturpuffer wartet.

Taste	Abfrage-code		ASCII oder Erweitert*			ASCII oder Erweitert* mit UMSCHALT			ASCII oder Erweitert* mit STRG			ASCII oder Erweitert* mit ALT		
	Dez	Hex	Dez	Hex	Zei	Dez	Hex	Zei	Dez	Hex	Zei	Dez	Hex	Zei
L SHIFT	42	2A												
\ |	43	2B	92	5C	\	124	7C	|	28	1C				
Z	44	2C	122	7A	z	90	5A	Z	26	1A		44	2C	NUL
X	45	2D	120	78	x	88	58	X	24	18		45	2D	NUL
C	46	2E	99	63	c	67	43	C	3	03		46	2E	NUL
V	47	2F	118	76	v	86	56	V	22	16		47	2F	NUL
B	48	30	98	62	b	66	42	B	2	02		48	30	NUL
N	49	31	110	6E	n	78	4E	N	14	0E		49	31	NUL
M	50	32	109	6D	m	77	4D	M	13	0D		50	32	NUL
, <	51	33	44	2C	,	60	3C	<						
. >	52	34	46	2E	.	62	3E	>						
/ ?	53	35	47	2F	/	63	3F	?						
R SHIFT	54	36												
* PRTSC	55	37	42	2A	*		INT 5**		16	10				
ALT	56	38												
SPACE	57	39	32	20	SPC	32	20	SPC	32	20	SPC	32	20	SPC
CAPS	58	3A												
F1	59	3B	59	3B	NUL	84	54	NUL	94	5E	NUL	104	68	NUL
F2	60	3C	60	3C	NUL	85	55	NUL	95	5F	NUL	105	69	NUL
F3	61	3D	61	3D	NUL	86	56	NUL	96	60	NUL	106	6A	NUL
F4	62	3E	62	3E	NUL	87	57	NUL	97	61	NUL	107	6B	NUL
F5	63	3F	63	3F	NUL	88	58	NUL	98	62	NUL	108	6C	NUL
F6	64	40	64	40	NUL	89	59	NUL	99	63	NUL	109	6D	NUL
F7	65	41	65	41	NUL	90	5A	NUL	100	64	NUL	110	6E	NUL
F8	66	42	66	46	NUL	91	5B	NUL	101	65	NUL	111	6F	NUL
F9	67	43	67	43	NUL	92	5C	NUL	102	66	NUL	112	70	NUL
F10	68	44	68	44	NUL	93	5D	NUL	103	67	NUL	113	71	NUL
NUM	69	45												
SCROLL	70	46												
HOME	71	47	71	47	NUL	55	37	7	119	77	NUL			
UP	72	48	72	48	NUL	56	38	8						
PGUP	73	49	73	49	NUL	57	39	9	132	84	NUL			
GREY -	74	4A	45	2D	-	45	2D	-						
LEFT	75	4B	75	4B	NUL	52	34	4	115	73	NUL			
CENTER	76	4C				53	35	5						
RIGHT	77	4D	77	4D	NUL	54	36	6	116	74	NUL			
GREY +	78	4E	43	2B	+	43	2B	+						
END	79	4F	79	4F	NUL	49	31	1	117	75	NUL			
DOWN	80	50	80	50	NUL	50	32	2						
PGDN	81	51	81	51	NUL	51	33	3	118	76	NUL			
INS	82	52	82	52	NUL	48	30	0						
DEL	83	53	83	53	NUL	46	2E	.						

** Unter DOS verursacht SHIFT+PRTSCR einen Interrupt-5, welcher den Bildschirminhalt druckt, bis ein Interrupt-Handler Ersetzung des Interrupt-5-Handlers definiert wurde.

D.2 ASCII-Zeichencodes

STRG	Dez	Hex	Zei	Code
^@	0	00		NUL
^A	1	01	☺	SOH
^B	2	02	☻	STX
^C	3	03	♥	ETX
^D	4	04	♦	EOT
^E	5	05	♣	ENQ
^F	6	06	♠	ACK
^G	7	07	•	BEL
^H	8	08	◘	BS
^I	9	09	○	HT
^J	10	0A	◙	LF
^K	11	0B	♂	VT
^L	12	0C	♀	FF
^M	13	0D	♪	CR
^N	14	0E	♫	SO
^O	15	0F	☼	SI
^P	16	10	►	DLE
^Q	17	11	◄	DC1
^R	18	12	↕	DC2
^S	19	13	‼	DC3
^T	20	14	¶	DC4
^U	21	15	§	NAK
^V	22	16	▬	SYN
^W	23	17	↨	ETB
^X	24	18	↑	CAN
^Y	25	19	↓	EM
^Z	26	1A	→	SUB
^[	27	1B	←	ESC
^\	28	1C	∟	FS
^]	29	1D	↔	GS
^^	30	1E	▲	RS
^_	31	1F	▼	US

Dez	Hex	Zei
32	20	
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	32	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dez	Hex	Zei
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dez	Hex	Zei
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	¦
125	7D	}
126	7E	~
127	7F	⌂*

* Der ASCII-Code 127 hat den Code DEL. Unter DOS hat dieser Code denselben Effekt wie ASCII 8 (BS). Der DEL-Code kann durch CTRL+BKSP erzeugt werden.

Dez	Hex	Zei	Dez	Hex	Zei	Dez	Hex	Zei	Dez	Hex	Zei
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	β
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	232	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	

Anhang E: Reservierte Wörter in QuickBASIC

Die folgende Liste enthält die von Microsoft BASIC reservierten Wörter:

ABS	CLS	ELSE	IF
ACCESS	COLOR	ELSEIF	IMP
ALIAS	COM	END	INKEY$
AND	COMMAND$	ENDIF	INP
ANY	COMMON	ENVIRON	INPUT
APPEND	CONST	ENVIRON$	INPUT$
AS	COS	EOF	INSTR
ASC	CSNG	EQV	INT
ATN	CSRLIN	ERASE	INTEGER
BASE	CVD	ERDEV	IOCTL
BEEP	CVDMBF	ERDEV$	IOCTL$
BINARY	CVI	ERL	IS
BLOAD	CVL	ERR	KEY
BSAVE	CVS	ERROR	KILL
BYVAL	CVSMBF	EXIT	LBOUND
CALL	DATA	EXP	LCASE$
CALLS	DATE$	FIELD	LEFT$
CASE	DECLARE	FILEATTR	LEN
CDBL	DEF	FILES	LET
CDECL	DEFDBL	FIX	LINE
CHAIN	DEFINT	FOR	LIST
CHDIR	DEFLNG	FRE	LOC
CHR$	DEFSNG	FREEFILE	LOCAL
CINT	DEFSTR	FUNCTION	LOCATE
CIRCLE	DIM	GET	LOCK
CLEAR	DO	GOSUB	LOF
CLNG	DOUBLE	GOTO	LOG
CLOSE	DRAW	HEX$	LONG

LOOP	**PEEK**	**SCREEN**	**TAB**
LPOS	**PEN**	**SEEK**	**TAN**
LPRINT	**PLAY**	**SEG**	**THEN**
LSET	**PMAP**	**SELECT**	**TIME$**
LTRIM$	**POINT**	**SETMEM**	**TIMER**
MID$	**POKE**	**SGN**	**TO**
MKD$	**POS**	**SHARED**	**TROFF**
MKDIR	**PRESET**	**SHELL**	**TRON**
MKDMBF$	**PRINT**	**SIGNAL**	**TYPE**
MKI$	**PSET**	**SIN**	**UBOUND**
MKL$	**PUT**	**SINGLE**	**UCASE$**
MKS$	**RANDOM**	**SLEEP**	**UEVENT**
MKSMBF$	**RANDOMIZE**	**SOUND**	**UNLOCK**
MOD	**READ**	**SPACE$**	**UNTIL**
NAME	**REDIM**	**SPC**	**USING**
NEXT	**REM**	**SQR**	**VAL**
NOT	**RESET**	**STATIC**	**VARPTR**
OCT$	**RESTORE**	**STEP**	**VARPTR$**
OFF	**RESUME**	**STICK**	**VARSEG**
ON	**RETURN**	**STOP**	**VIEW**
OPEN	**RIGHT$**	**STR$**	**WAIT**
OPTION	**RMDIR**	**STRIG**	**WEND**
OR	**RND**	**STRING**	**WHILE**
OUT	**RSET**	**STRING$**	**WIDTH**
OUTPUT	**RTRIM$**	**SUB**	**WINDOW**
PAINT	**RUN**	**SWAP**	**WRITE**
PALETTE	**SADD**	**SYSTEM**	**XOR**
PCOPY			

Anhang F: Metabefehle

Dieser Anhang beschreibt die QuickBASIC – Metabefehle – Befehle, die QuickBASIC anweisen, das Programm in einer ganz bestimmten Art und Weise zu behandeln. Der erste Abschnitt beschreibt das für Metabefehle verwendete Format. Die nächsten zwei Abschnitte stellen bestimmte Metabefehle vor.

Mit Hilfe der Metabefehle können Sie:

- Andere BASIC-Quelldateien an bestimmten Stellen während der Kompilierung einlesen und kompilieren (**$INCLUDE**).
- Die Zuordnung dimensionierter Datenfelder steuern (**$STATIC** und **$DYNAMIC**).

F.1 Die Metabefehlssyntax

Metabefehle beginnen mit einem Dollarzeichen (**$**) und sind immer in einen Programmkommentar eingeschlossen. In einem Kommentar kann mehr als ein Metabefehl stehen. Mehrere Metabefehle werden durch Leerraumzeichen – Leerzeichen oder Tabulatoren – voneinander getrennt. Bei Metabefehlen, denen Argumente folgen, steht zwischen dem Befehl und dem Argument ein Doppelpunkt:

REM *$Metabefehl [: Argument]*

Zeichenkettenargumente müssen in einfache Anführungszeichen gesetzt werden. Leerraumzeichen zwischen den Elementen eines Metabefehls werden übergangen. Nachstehend werden die gültigen Metabefehlsformate aufgelistet:

```
REM $STATIC $INCLUDE:  'datendef.bi'
REM     $STATIC    $INCLUDE :  'datendef.bi'
' $STATIC $INCLUDE:  'datendef.bi'
'     $STATIC    $INCLUDE :  'datendef.bi'
```

Bitte beachten Sie, daß zwischen dem Dollarzeichen und dem Rest des Metabefehls kein Leerzeichen stehen darf.

Wenn Sie sich in einer Beschreibung auf einen Metabefehl beziehen möchten, der nicht ausgeführt werden soll, geben Sie vor dem ersten Dollarzeichen in der Zeile ein beliebiges Zeichen, mit Ausnahme des Tabulators oder des Leerzeichens, ein. Beispielsweise werden in der folgenden Zeile beide Metabefehle übergangen:

```
REM x$STATIC $INCLUDE:   'datendef.bi'
```

F.2 Verarbeitung zusätzlicher Quelldateien: $INCLUDE

Der Metabefehl **$INCLUDE** weist den Compiler an, vorübergehend die Verarbeitung einer Datei zu verlassen und stattdessen Programmanweisungen der im Argument angegebenen BASIC-Datei zu lesen. Wenn das Ende der eingefügten Datei erreicht wird, setzt der Compiler die Verarbeitung der ursprünglichen Datei fort. Da die Kompilierung mit der Zeile beginnt, die unmittelbar auf die Zeile mit dem Befehl **$INCLUDE** folgt, sollte die Anweisung **$INCLUDE** am Ende einer Zeile stehen. Die nachstehende Anweisung ist richtig:

```
DEFINT I-N   ' $INCLUDE:  'COMMON.BAS'
```

Die beiden folgenden Einschränkungen sind bei der Benutzung von Include-Dateien zu beachten:

1. Eingefügte Dateien dürfen keine **SUB**- oder **FUNCTION**-Anweisungen enthalten.
2. Eingefügte mit BASICA erstellte Dateien müssen mit der Option ,A gespeichert werden.

F.3 Zuordnung von dimensionierten Datenfeldern: $STATIC und $DYNAMIC

Die **$STATIC**- und **$DYNAMIC**-Metabefehle teilen dem Compiler die Zuordnung von Speicherplatz für Datenfelder mit. Keiner dieser Metabefehle nimmt ein Argument an:

```
'Mache alle Datenfelder dynamisch.
'$DYNAMIC
```

$STATIC reserviert Speicherplatz für Datenfelder während der Kompilierung. Wenn **$STATIC** benutzt wird, reinitialisiert die **ERASE**-Anweisung alle Datenfeldwerte mit Null (numerische Datenfelder) oder mit der Null-Zeichenkette (Zeichenketten-Datenfelder), ohne die Datenfelder zu entfernen. **REDIM** hat keine Auswirkungen auf **$STATIC**-Datenfelder.

$DYNAMIC ordnet Datenfeldern Speicherplatz während des Programmlaufs zu. Das bedeutet, daß die **ERASE**-Anweisung die Datenfelder entfernt und den beanspruchten Speicherplatz für andere Zwecke freigibt. Die Größe eines **$DYNAMIC**-Datenfeldes läßt sich auch mit Hilfe der Anweisung **REDIM** verändern.

Die **$STATIC**- und **$DYNAMIC**-Metabefehle wirken sich auf alle Datenfelder, mit Ausnahme implizit dimensionierter Datenfelder aus (in einer **DIM**-Anweisung nicht deklarierte Datenfelder). Implizit dimensionierte Datenfelder werden stets wie mit **$STATIC** zugeordnet.

Anhang G: Kompilieren und Binden aus DOS

Dieser Anhang erläutert das Kompilieren und Binden außerhalb der QuickBASIC-Umgebung. Dieses Verfahren kann aus folgenden Gründen zweckmäßig sein:

- Verwendung eines anderen Texteditors.
- Erstellung ausführbarer Programme, deren Fehler mit Hilfe des Debuggers Microsoft CodeView beseitigt werden können.
- Erstellung von Listendateien, die zur Beseitigung von Fehlern in einem selbständig ausführbaren Programm dienen.
- Verwendung von Optionen, die innerhalb der QuickBASIC-Umgebung nicht verfügbar sind, wie zum Beispiel zeilenweise gespeicherte Datenfelder.
- Binden mit der von QuickBASIC gelieferten Datei *noem.obj*, die die Größe der ausführbaren Dateien in Programmen reduziert, die immer mit einem mathematischen Koprozessor verwendet werden.

Wenn Sie diesen Anhang gelesen haben, können Sie:

- Aus der DOS-Befehlszeile anhand des BC-Befehls kompilieren.
- Ausführbare Dateien erstellen und Programm-Objektdateien mit Hilfe des Befehls LINK binden.
- Selbständige Bibliotheken (*.lib*) mit Hilfe des Befehls LIB erstellen und pflegen.

G.1 BC, LINK und LIB

Das Microsoft QuickBASIC-Paket enthält auch die Befehle BC, LINK und LIB. Folgende Liste beschreibt die Verwendung dieser Dienstprogramme beim Kompilieren und Binden außerhalb der QuickBASIC-Umgebung:

Programm	***Funktion***
bc.exe	Bei der Auswahl der Befehle **EXE-Datei erstellen** oder Bibliothek erstellen aus dem Menü **Ausführen**, ruft QuickBASIC den BASIC-Befehlszeilen-Compiler (BC) auf, um die als Objektdateien bezeichneten Zwischendateien zu erstellen. Diese werden zusammengebunden und bilden das Programm oder die Quick-Bibliothek. BC steht jedoch auch zum Kompilieren von Programmen außerhalb der QuickBASIC-Umgebung zur Verfügung. BC ist eventuell bei der Kompilierung eines Programms, das mit einem anderen Texteditor geschrieben wurde, zu bevorzugen. BC braucht jedoch nur dann aus der Befehlszeile verwendet zu werden, wenn das Programm für die Kompilierung im Speicher innerhalb der QuickBASIC-Umgebung zu umfangreich ist, oder wenn die ausführbare Datei zu dem CodeView-Debugger kompatibel sein soll.
link.exe	QuickBASIC verwendet den Microsoft Overlay Linker (LINK), um von BC erstellte Objektdateien mit den entsprechenden Bibliotheken zu binden und eine ausführbare Datei zu erzeugen. LINK kann jederzeit zur Bindung von Objektdateien und Erstellung von Quick-Bibliotheken benutzt werden.
lib.exe	Der Microsoft-Bibliotheksmanager (LIB) erstellt selbständige Bibliotheken aus Objektdateien, die anhand von BC angelegt wurden. QuickBASIC selbst erstellt solche Bibliotheken anhand von LIB und verwendet diese anschließend, wenn der Befehl **EXE-Datei erstellen** aus dem Menü **Ausführen** gewählt wird.

G.2 Der Kompilier- und Bindeprozeß

Zur Erstellung eines selbständigen Programmes aus einer BASIC-Quelldatei außerhalb der QuickBASIC-Umgebung, verfahren Sie wie folgt:

1. Kompilieren Sie jede Quelldatei, um eine Objektdatei zu erstellen.

2. Binden Sie die Objektdateien mit Hilfe von LINK. LINK bindet eine oder mehrere selbständige Bibliotheken ein und legt eine ausführbare Datei an. Zunächst stellt LINK jedoch sicher, daß alle Prozeduraufrufe in den Quelldateien mit den Prozeduren in den Bibliotheken oder in anderen Objektdateien übereinstimmen.

Sie können mit Hilfe einer der beiden folgenden Methoden kompilieren und binden:

- Sie können anhand der Befehle BC und LINK schrittweise kompilieren und binden.
- Sie können eine Stapelverarbeitungsdatei anlegen, die alle Kompilier- und Bindebefehle enthält. Diese Methode ist besonders hilfreich, wenn Sie beim Kompilieren und Binden der Programme immer dieselben Optionen verwenden.

Hinweis Wenn QuickBASIC das Programm innerhalb der Umgebung kompiliert und bindet, wird die Linker-Option /E automatisch gesetzt. Beim Einsatz des Befehls LINK außerhalb der QuickBASIC-Umgebung, muß jedoch die Option /E ausdrücklich angegeben werden, um die Größe der ausführbaren Datei zu minimieren und die Ladegeschwindigkeit des Programms zu maximieren.

Beim Kompilieren und Binden aus DOS kommen die im Menü **Optionen** definierten Pfade nicht zur Anwendung. Um nach Include- und Bibliothekdateien auf die im Menü **Optionen** angegebene Weise zu suchen, müssen die DOS-Umgebungsvariablen LIB und INCLUDE zur Angabe der geeigneten Verzeichnisse gesetzt werden. Andernfalls kann es vorkommen, daß der Compiler und/oder der Linker Fehler des Typs `Datei nicht gefunden` erzeugen.

Die Abschnitte G.3 und G.4 erläutern das schrittweise Kompilieren und Binden.

G.3 Kompilieren mit Hilfe des Befehls BC

Sie können mit dem Befehl BC auf eine der folgenden beiden Arten kompilieren:

1. Geben Sie alle Informationen auf einer einzelnen Befehlszeile anhand folgender Syntax ein:

   BC *Quelldatei* [, [*Objektdatei*] [, [*Listendatei*]]][*Optionsliste*][;]

2. Geben Sie

   ```
   BC
   ```

   ein und beantworten Sie folgende Anfragen:

   ```
   Quelldatei [.bas]:
   Objektdatei [Basisname.obj]:
   Quell-Listing: [NUL.LST]:
   ```

Tabelle G.1 zeigt die Eingabe, die in der BC-Befehlszeile oder als Antwort auf jede Anfrage einzusetzen ist:

Tabelle G.1 Eingabe auf den Befehl BC

Feld	***Anfrage***	***Eingabe***
Quelldatei	"Quelldatei"	Name der jeweiligen Quelldatei.
Objektdatei	"Objektdatei"	Name der erstellten Objektdatei.
Listendatei	"Quell-Listing"	Name der Datei, die ein Quell-Listing enthält. Die Quell-Listingdatei enthält die Adresse jeder Zeile der Quelldatei, deren Text und Größe sowie jede während der Kompilierung erzeugte Fehlermeldung.
Optionsliste	Bietet Optionen nach jeder Antwort	Jede der in Abschnitt G.3.2, "Verwendung der BC-Befehlsoptionen" beschriebenen Compiler-Optionen.

G.3.1 Angabe der Dateinamen

Der BC-Befehl geht aufgrund der für die Dateien verwendeten Pfadnamen und Erweiterungen von bestimmten Voraussetzungen hinsichtlich der angegebenen Dateien aus. Die folgenden Abschnitte beschreiben diese Voraussetzungen und andere Regeln, die bei der Angabe von Dateinamen für den Befehl BC befolgt werden sollten.

G.3.1.1 Groß- und Kleinbuchstaben

Für Dateinamen kann jede beliebige Kombination von Groß- und Kleinbuchstaben verwendet werden; der Compiler betrachtet Klein- und Großbuchstaben als austauschbar.

Beispiel

Für den Befehl BC sind folgende drei Dateinamen identisch:

```
abcde.BAS
ABCDE.BAS
aBcDe.Bas
```

G.3.1.2 Erweiterungen für Dateinamen

Ein DOS-Dateiname besteht aus zwei Teilen: dem "Basisnamen", der alle Zeichen bis zum Punkt (.) umfaßt (ohne diesen zu enthalten) sowie der "Erweiterung", die den Punkt und bis zu drei darauffolgende Zeichen, enthält. Im allgemeinen kennzeichnet die Erweiterung den Typ der Datei (zum Beispiel, ob es sich um eine BASIC-Quelldatei, eine Objektdatei, eine ausführbare Datei oder eine selbständige Bibliothek handelt).

BC und LINK verwenden für Dateinamen die in der folgenden Liste erläuterten Erweiterungen:

Erweiterung	***Erklärung***
.bas	BASIC-Quelldatei
.obj	Objektdatei
.lib	Selbständige Bibliotheksdatei
.lst	Die von BC erstellte Listendatei
.map	Datei für Symbole des gebundenen Programms
.exe	Ausführbare Datei

G.3.1.3 Pfadnamen

Jeder Dateiname kann einen vollständigen oder teilweisen Pfadnamen enthalten. Ein vollständiger Pfadname beginnt mit der Laufwerksangabe; ein Teilpfadname besteht aus einem oder mehreren Verzeichnisnamen vor dem Dateinamen, enthält jedoch keine Laufwerksangabe.

Ein vollständiger Pfadname erlaubt die Angabe von Dateien mit unterschiedlichen Pfaden als Eingabe für den Befehl BC und ermöglicht die Erstellung von Dateien in unterschiedlichen Laufwerken oder Verzeichnissen des aktuellen Laufwerks.

Hinweis Für Dateien, die mit Hilfe von BC angelegt werden, kann ein Pfadname angegeben werden, der mit einem rückwärtigen Schrägstrich (\) endet, um die Datei in diesem Pfad anzulegen. Bei der Erstellung der Datei verwendet BC deren Standardnamen.

G.3.2 Verwendung der BC-Befehlsoptionen

Optionen des Befehls BC bestehen entweder aus einem Schrägstrich (/) oder aus einem Bindestrich (-), denen ein oder mehrere Buchstaben folgen. (Der Schräg- und der Bindestrich sind in diesem Fall austauschbar. In diesem Handbuch werden Schrägstriche für Optionen verwendet.)

Die BC-Befehlszeilen-Optionen werden in der folgenden Liste erläutert:

Option	***Beschreibung***
/A	Erzeugt ein Listing mit disassembliertem Objektcode für jede Quellzeile und zeigt den vom Compiler erzeugten Assembler-Code.
/AH	Ermöglicht es dynamischen Feldern, welche Datensätze, Zeichenketten fester Länge oder numerische Daten enthalten, den gesamten verfügbaren Speicher einzunehmen. Wenn diese Option nicht angegeben ist, beträgt die maximale Größe pro Datenfeld 64K. Beachten Sie, daß sich diese Option auf die Übergabe von Daten an Prozeduren nicht auswirkt.
/C:*Puffer-größe*	Setzt für jeden Kommunikationsanschluß die Größe des Puffers, der Ferndaten über einen asynchronen Kommunikationsadapter empfängt. (Dem Übertragungspuffer wird für jeden Übertragungsanschluß eine Kapazität von 128 Bytes zugewiesen; der Puffer kann nicht auf der BC-Befehlszeile verändert werden.) Diese Option hat keine Auswirkungen, falls keine asynchrone Datenübertragungskarte vorhanden ist. Die Standard-Puffergröße beträgt für beide Anschlüsse insgesamt 512 Bytes; die maximale Größe beträgt 32.767 Bytes.
/D	Erzeugt Debug-Code für die Laufzeit-Fehlerprüfung und aktiviert STRG+UNTBR. Diese Option ist identisch mit der Option **Debug-Code erstellen** des Befehls **EXE-Datei erstellen** aus dem Menü **Ausführen** innerhalb der QuickBASIC-Umgebung.
/E	Zeigt das Vorhandensein von **ON ERROR**- mit **RESUME** *Zeilennummer*-Anweisungen an. (Siehe auch die Erläuterung der Option /X in dieser Liste.)
/MBF	Die eingebauten Funktionen **MKS$**, **MKD$**, **CVS** und **CVD** werden jeweils übertragen in **MKSMBF$**, **MKDMBF$**, **CVSMBF** und **CVDMBF**. Diese Konversion erlaubt dem QuickBASIC-Programm das Lesen und Schreiben der im Microsoft Binärformat gespeicherten Gleitkommawerte.
/O	Ersetzt *brun45.lib* durch die Laufzeit-Bibliothek *bcom45.lib*. Weitere Informationen zur Verwendung dieser Bibliotheken finden Sie in Kapitel 16, "Das Menü Ausführen", des Handbuches *Lernen und Anwenden von Microsoft QuickBASIC*.
/R	Speichert Datenfelder zeilenweise. Normalerweise speichert BASIC Datenfelder spaltenweise. Diese Option ist beim Einsatz anderssprachiger Routinen behilflich, die Datenfelder zeilenweise speichern.
/S	Schreibt in Anführungszeichen gesetzte Zeichenketten in die Objektdatei, nicht in die Symboltabelle. Diese Option ist beim Auftreten der Fehlermeldung `Speicherkapazität reicht nicht aus` in einem Programm zu verwenden, in dem zahlreiche Zeichenkettenkonstanten vorkommen.

Option	*Beschreibung*
/V	Schaltet die Ereignisverfolgung für Datenübertragung (**COM**), den Lichtstift (**PEN**), den Joystick (**STRIG**), die Uhr (**TIMER**), den Musikpuffer (**PLAY**) und die Funktionstasten (**KEY**) ein. Diese Option ist zur Überprüfung der zwischen Anweisungen auftretenden Ereignisse zu verwenden.
/W	Schaltet die Ereignisverfolgung für dieselben Anweisungen wie /V ein, prüft jedoch zwischen jeder Zeilennummer oder Zeilenmarke auf das Auftreten eines Ereignisses.
/X	Kennzeichnet das Vorhandensein von **ON ERROR** mit **RESUME**, **RESUME NEXT** oder **RESUME 0**.
/ZD	Erzeugt eine Objektdatei, die Datensätze mit Zeilennummern enthält, die den Zeilennummern der Quelldatei entsprechen. Diese Option ist bei der Fehlerbeseitigung auf Quellebene mit Hilfe der Microsoft "Symbolic Debug Utility" (SYMDEB), die mit der Microsoft Macro Assembler Version 4.0 verfügbar ist, behilflich.
/ZI	Erzeugt eine Objektdatei, die Debug-Informationen enthält und vom Microsoft CodeView-Debugger verwendet wird, der mit Microsoft C Version 5.0 oder spätere und dem Microsoft Macro Assembler Version 5.0 oder spätere verfügbar ist.

G.4 Binden

Nach Kompilierung des Programms muß die Objektdatei mit den passenden Bibliotheken gebunden werden, um ein ausführbares Programm zu erzeugen. Der LINK-Befehl kann wie folgt benutzt werden:

- Geben Sie auf der Befehlszeile folgendes ein:

  LINK *Objdatei* [,[*Exedatei*][,[*Mapdatei*][,[*Bibl*]]][*Linkoptionen*][;]]

  Die Befehlszeile darf nicht mehr als 128 Zeichen enthalten.

- Schreiben Sie

  ```
  LINK
  ```

  und beantworten Sie folgende Anfragen:

  ```
  Objektmodule [obj]:
  Ausführbare Datei [Basisname.exe]:
  List-Datei [NUL.MAP]:
  Bibliotheken [.lib]:
  ```

  Um nach einer Anfrage mehrere Dateien anzugeben, geben Sie am Ende der Zeile ein Pluszeichen (+) ein. Die Anfrage erscheint erneut auf der nächsten Zeile, und Sie können mit der Eingabe für die Anfrage fortfahren.

- Richten Sie eine "Antwortdatei" ein, die Antworten auf die LINK-Befehlsanfragen enthält, und geben Sie anschließend einen LINK-Befehl folgender Form ein:

  `LINK @` *Dateiname*

  An dieser Stelle ist *Dateiname* der Name der Antwortdatei. Jeder Antwort können Sie Linker-Optionen hinzufügen oder auf einer oder mehreren separaten Zeilen Optionen angeben. Die Antworten müssen in derselben Reihenfolge wie die oben erläuterten LINK-Befehlsanfragen vorliegen. Sie können auch den Namen einer Antwortdatei nach jeder Linker-Anfrage oder an beliebiger Stelle auf der LINK-Befehlszeile eingeben.

  Tabelle G.2 zeigt die Eingaben, die auf der LINK-Befehlszeile oder als Anwort auf jede Anfrage angegeben werden müssen.

Tabelle G.2 Eingabe auf den Befehl LINK

Feld	***Anfrage***	***Eingabe***
Objdatei	"Objektmodule"	Eine oder mehrere zu bindende Objektdateien. Die Objektdateien sollten durch Pluszeichen oder Leerzeichen voneinander getrennt sein.
Exedatei	"Ausführbare Datei"	Name der erstellten ausführbaren Datei, wenn Sie dieser einen Namen oder eine Erweiterung geben wollen, die vom Standard abweichen. Es ist stets die Erweiterung *.exe* zu verwenden, da DOS diese Erweiterung bei den ausführbaren Dateien erwartet.
Mapdatei	"List-Datei"	Name der Datei, die eine eventuell angelegte Symboltabelle enthält.* Es kann auch einer der folgenden DOS-Gerätenamen zur Umleitung der MAP-Datei auf dieses Gerät angegeben werden: AUX für ein Zusatzgerät, CON für die Konsole (Terminal), PRN für einen Drucker oder NUL für kein Gerät (so daß keine Map-Datei angelegt wird). Ein Beispiel für eine Map-Datei sowie Informationen über deren Inhalt finden Sie in Abschnitt G.4.6.11.

Feld	***Anfrage***	***Eingabe***
Bibl	"Bibliotheken"	Eine oder mehrere selbständige Bibliotheken (oder Verzeichnisse, die nach selbständigen Bibliotheken durchsucht werden sollen), die durch Pluszeichen oder Leerzeichen getrennt sind. Die Anfrage "Bibliotheken" erlaubt die Angabe von höchstens 16 Bibliotheken; jede weitere Bibliotheksangabe wird ignoriert. In Abschnitt G.4.3 finden Sie die Regeln für die Angabe der Bibliotheksnamen an den Linker.
Linkoptionen	Gibt Optionen nach jeder Anwort	Jede der in den Abschnitten G.4.6.2 bis G.4.6.15 beschriebenen Linker-Optionen, die an beliebiger Stelle auf der Befehlszeile angegeben werden können.

* Eine andere Möglichkeit, eine Map-Datei zu erstellen, besteht in der Angabe der Option /MAP für den Befehl LINK (siehe Abschnitt G.4.6.11).

Falls Sie eine Antwortdatei verwenden, muß jede Antwort den in der obigen Tabelle beschriebenen Regeln entsprechen.

G.4.1 Vorgabewerte für LINK

Sie können Vorgabewerte für alle vom Linker benötigten Informationen wie folgt wählen:

- Um den Vorgabewert für einen Befehlszeileneintrag zu wählen, lassen Sie den bzw. die Dateinamen vor dem Eintrag aus und geben nur das erforderliche Komma ein. Die einzige Ausnahme ist der Vorgabewert für den Eintrag der Map-Datei: Wird ein Komma als Platzhalter für diesen Eintrag eingesetzt, erzeugt der Linker eine Map-Datei.
- Um den Vorgabewert für eine Anfrage zu wählen, betätigen Sie einfach die EINGABETASTE.
- Um die Vorgabewerte für alle verbleibenden Befehlszeileneinträge oder -anfragen zu wählen, geben Sie nach jedem Eintrag bzw. nach jeder Anfrage ein Semikolon ein. Die einzigen erforderlichen Angaben sind ein oder mehrere Namen von Objektdateien.

Die folgende Liste zeigt die vom Linker für ausführbare Dateien, Map-Dateien sowie Bibliotheken verwendeten Vorgabewerte:

Dateityp	***Vorgabewert***
Ausführbar	Der Basisname der ersten angegebenen Objektdatei und die Erweiterung *.exe*. Um die ausführbare Datei umzubenennen, ist lediglich der neue Basisname anzugeben; wird ein Dateiname ohne Erweiterung angegeben, fügt der Linker automatisch die Erweiterung *.exe* an.
Map	Der besondere Dateiname *nul.map*, der dem Linker mitteilt, *keine* Map-Datei zu erzeugen. Um eine Map-Datei anzulegen, ist nur der Basisname anzugeben; wird ein Dateiname ohne Erweiterung angegeben, fügt der Linker automatisch die Erweiterung *.map* an.
Bibliotheken	Bibliotheken, die in den angegebenen Objektdateien genannt sind. Bei der Auswahl der Option **Selbständige EXE-Datei**, ist die Standard-Bibliothek *bcom45.lib*. Andernfalls ist die Standard-Bibliothek *brun40.lib*. Falls Sie eine andere als die Standard-Bibliothek angeben, brauchen Sie nur den Basisnamen einzugeben; nach Eingabe eines Bibliotheknamens ohne Erweiterung fügt der Linker automatisch die Erweiterung *.lib* hinzu. Informationen über die Angabe anderer als der Standard-Bibliotheken finden Sie in Abschnitt G.4.3.

Beispiele

Das folgende Beispiel veranschaulicht eine Antwortdatei. Sie weist den Linker an, die vier Objektmodule `RAHMEN, TEXT, TABELLE` und `SKIZZE` zu binden. Es werden sowohl die als *rahmen.exe* bezeichnete ausführbare Datei als auch die als *rahmensy.map* bezeichnete Map-Datei erzeugt. Die Option /PAUSE veranlaßt den Linker vor Erstellung der ausführbaren Datei anzuhalten, um einen eventuell notwendigen Diskettenwechsel zu erlauben. Die Option /MAP teilt dem Linker mit, globale Symbole und Adressen in die Map-Datei einzufügen. Der Linker bindet außerdem jede aus der Bibliotheksdatei *graf.lib* benötigte Routine ein. Weitere Informationen zu den Optionen /PAUSE und /MAP finden Sie in den Abschnitten G.4.6.2 und G.4.6.11.

```
RAHMEN TEXT TABELLE SKIZZE
/PAUSE /MAP
RAHMENSY
GRAF.lib
```

Im folgenden Beispiel lädt und bindet der Linker die Objektdateien *rahmen.obj*, *text.obj*, *tabelle.obj* und *skizze.obj* und durchsucht die Bibliotheksdatei *hilflib.lib* nach ungelösten Bezügen. Die ausführbare Datei wird standardmäßig *rahmen.exe* genannt. Außerdem wird eine Map-Datei erzeugt, die *rahmensy.map* heißt.

```
LINK RAHMEN+TEXT+TABELLE+SKIZZE, ,RAHMENSY, HILFLIB.LIB
```

Das folgende Beispiel verdeutlicht, wie eine Anfrage mit der Eingabe eines Pluszeichens (+) am Ende der Antwort fortgesetzt wird. Das Beispiel bindet alle angegebenen Objektdateien und erstellt anschließend eine ausführbare Datei. Da als Antwort auf die Anfrage "Ausführbare Datei" ein Semikolon eingegeben wurde, wird die ausführbare Datei standardmäßig mit dem Basisnamen der ersten angegebenen Objektdatei (RAHMEN) und der Erweiterung *.exe* benannt. Die Vorgabewerte werden auch für die verbleibenden Anfragen verwendet; daher wird keine Map-Datei angelegt und für das Binden werden die in den Objektdateien bezeichneten Standard-Bibliotheken verwendet.

```
LINK
Objektmodule [.OBJ]: RAHMEN TEXT TABELLE SKIZZE+
Objektmodule [.OBJ]: BASIS ZURUECK SPALNUM+
Objektmodule [.OBJ]: ZEILNUM
Ausführbare Datei [RAHMEN.EXE]: ;
```

G.4.2 Dateiangabe für LINK

Die Regeln zur Angabe von Dateinamen für den Linker sind dieselben wie die Regeln zur Angabe von Dateinamen für den Befehl BC: Groß- und Kleinbuchstaben können beliebig verwendet werden und Dateinamen können Pfadnamen enthalten, die dem Linker mitteilen, in dem angegebenen Pfad nach Dateien zu suchen bzw. Dateien anzulegen. Weitere Informationen finden Sie in Abschnitt G.3.1.

G.4.3 Angabe von Bibliotheken für LINK

Normalerweise brauchen Sie dem Linker den Namen einer selbständigen Bibliothek nicht anzugeben. Beim Erstellen der Objektdateien fügt der Befehl BC in jeder Objektdatei den Namen der richtigen selbständigen Bibliothek für diese Objektdatei ein. Bei Übertragung der Objektdatei sucht der Linker nach einer Bibliothek mit dem Namen, der in der Objektdatei steht, und bindet die Objektdatei automatisch mit dieser Bibliothek.

Um Objektdateien mit einer anderen selbständigen Bibliothek als der Standard-Bibliothek zu binden, teilen Sie dem Linker den Namen der Nicht-Standard-Bibliothek mit. Sie können den Bibliotheksnamen wie folgt eingeben:

- Nach dem dritten Komma auf der LINK-Befehlszeile. Kommata stehen nach der Liste von Objektdateinamen, dem Namen der ausführbaren Datei sowie dem Namen der List-Datei. Der letzte Name entspricht dem Bibliotheksnamen.
- Als Antwort auf die Anfrage "Bibliotheken" des LINK-Befehls.

Vor Durchsuchen der Standard-Bibliotheken durchsucht der Linker die angegebenen Bibliotheken, um externe Bezüge aufzulösen.

Es kann vorkommen, daß Sie aus einem der folgenden Gründe mit einer anderen selbständigen Bibliothek als der Standard-Bibliothek binden möchten:

- Binden mit zusätzlichen selbständigen Bibliotheken.
- Binden mit Bibliotheken in unterschiedlichen Pfaden. Wenn Sie für die Bibliothek einen kompletten Pfadnamen angeben, sucht der Linker nur in diesem Pfad nach der Bibliothek. Andernfalls sucht er an einer der folgenden drei Stellen:
  1. In dem aktuellen Arbeitsverzeichnis.
  2. In jedem Pfad oder Laufwerk, der/das nach dem dritten Komma auf der Befehlszeile angegeben ist.
  3. An den von der LIB-Umgebungsvariablen angegebenen Stellen.
- Ignorieren der in der Objektdatei bezeichneten Bibliothek. In diesem Fall muß die LINK-Option /NOD zusätzlich zu der zu verwendenden Bibliothek angegeben werden. Weitere Informationen zu der Option **/NOD** finden Sie in Abschnitt G.4.6.8.

G.4.4 Speicheranforderungen des Linkers

Der Linker verwendet für den Bindevorgang den verfügbaren Speicher. Falls die zu bindenden Dateien eine Ausgabedatei bilden, die den verfügbaren Speicherbereich überschreitet, legt der Linker eine als Speicher dienende temporäre Diskettendatei an. Diese temporäre Datei wird je nach DOS-Version wie folgt behandelt:

- Der Linker verwendet das von der DOS-Umgebungsvariablen TMP angegebene Verzeichnis aus DOS zur Erstellung einer temporären Datei. Falls die Variable TMP zum Beispiel auf C:\TEMPDIR gesetzt wäre, würde der Linker die temporäre Datei in C:\TEMPDIR schreiben.

  Wenn keine TMP-Umgebungsvariable vorhanden ist, oder das von TMP angegebene Verzeichnis nicht existiert, schreibt der Linker die temporäre Datei in das aktuelle Arbeitsverzeichnis.

- Wenn der Linker mit DOS-Version 3.0 oder einer späteren Version läuft, verwendet er einen DOS-Systemaufruf zur Erstellung einer temporären Datei mit einem in dem Verzeichnis der temporären Datei eindeutigen Namen.
- Wenn der Linker mit einer DOS-Version kleiner als 3.0 läuft, erzeugt er eine temporäre Datei mit dem Namen VM.TMP.

Wenn der Linker eine temporäre Diskettendatei erstellt, erscheint die Meldung

```
Temporäre Datei Tempdatei ist angelegt.
Diskette in Laufwerk Buchstabe nicht wechseln
```

wobei *Tempdatei* ".\" von dem Dateinamen *vm.tmp* oder einem von DOS erzeugten Namen gefolgt wird und *Buchstabe* das Laufwerk, das die temporäre Datei enthält, bezeichnet. Die Meldung

```
Diskette in Laufwerk Buchstabe nicht wechseln
```

erscheint nur, wenn das mit *Buchstabe* bezeichnete Laufwerk eine Diskettenstation ist. Falls diese Meldung erscheint, entfernen Sie die Diskette bis zum Beenden des Bindevorgangs nicht aus dem Laufwerk. Beim Entfernen der Diskette verhält sich der Linker unvorhersehbar, und es könnte folgende Meldung erscheinen:

```
Unerwartetes Dateiende in temporärer Datei
```

Wenn Sie diese Meldung sehen, beginnen Sie erneut mit dem Binden.

Die von LINK angelegte temporäre Datei ist nur eine Arbeitsdatei. Nach Abschluß des Bindevorgangs wird sie vom Linker gelöscht.

Hinweis Geben Sie keiner Ihrer eigenen Dateien den Namen *vm.tmp*. Der Linker gibt eine Fehlermeldung aus, wenn er eine existierende Datei mit diesem Namen findet.

G.4.5 Binden anhand von Programmen in verschiedenen Sprachen

Programme in verschiedenen Sprachen können zwar anhand von LINK gebunden werden, aber das Binden von *.obj*-Dateien aus einer anderen Sprache kann zu Problemen führen. Unterschiedliche Voraussetzungen verschiedener Linker können QuickBASIC-Dateien zerstören.

Die folgenden Abschnitte behandeln das Binden mit Modulen, die in Pascal, FORTRAN und Assembliersprache geschrieben sind.

G.4.5.1 Pascal- und FORTRAN-Module in QuickBASIC-Programmen

Die mit Microsoft Pascal oder FORTRAN kompilierten Module lassen sich anhand von BASIC-Programmen binden, wie im *Microsoft Mixed Language Programming Guide* beschrieben. Diese Module lassen sich auch in Quick-Bibliotheken eingliedern. QuickBASIC-Programme, die mit Microsoft Pascal kompilierten Code enhalten, müssen jedoch mindestens 2K near-heap Speicherplatz für Pascal zuordnen. Im folgenden Beispiel ordnet die Anweisung **DIM** ein statisches Datenfeld von 2K oder größer einem gegebenen gemeinsamen Block mit dem Namen NMALLOC zu:

```
DIM Name%(2048) : COMMON SHARED /NMALLOC/ Name%()
```

Das Pascal Laufzeitmodul erwartet jederzeit einen verfügbaren Near-Heap-Speicherplatz von mindestens 2K. Falls der Pascal-Code den erforderlichen Speicherplatz nicht zuweisen kann, kann das einen Programmabsturz von QuickBASIC zur Folge haben. Dasselbe gilt sowohl für den Pascal-Code in Quick-Bibliotheken als auch für den in ausführbaren Dateien eingebundenen Pascal-Code. FORTRAN E/A erfordert ebenfalls einen nahen Pufferspeicherplatz (near buffer space), der von einem gemeinsamen Block NMALLOC zur Verfügung gestellt wird.

G.4.5.2 Zuordnung von STATIC-Datenfeldern in Routinen der Assembliersprache

Statische Datenfelder können mit Hilfe der Schlüsselwörter **SEG** oder **CALLS** sowie mit langen Zeigern an Routinen der Assembliersprache übergeben werden. Sie dürfen nicht von der Annahme ausgehen, daß sich die Daten in einem bestimmten Segment befinden. Es ist ebenfalls möglich, sämtliche Datenfelder unter der Verwendung langer Zeiger als dynamisch zu deklarieren, da BC und die QuickBASIC-Umgebung dynamische Datenfelder identisch behandeln.

G.4.5.3 Bezugnahme auf DGROUP in erweiterten Laufzeitmodulen

Vergewissern Sie sich bei Programmen in verschiedenen Sprachen, die den Befehl **CHAIN** benutzen, daß der in einem erweiterten Laufzeitmodul eingebaute Code sich nicht auf DGROUP bezieht. (Der Befehl **CHAIN** veranlaßt die Bewegung von DGROUP, ohne den Bezug auf DGROUP zu aktualisieren.) Diese Regel gilt nur für Programme in verschiedenen Sprachen; da sich BASIC-Routinen nicht auf DGROUP beziehen, ist keine solche Vorsicht bei Programmen geboten, die ausschließlich in BASIC geschrieben sind.

Dieses Problem läßt sich durch Verwendung des SS-Wertes vermeiden, da BASIC immer von der Übereinstimmung zwischen SS und DGROUP ausgeht.

G.4.6 Verwendung der LINK-Optionen

Alle LINK-Optionen beginnen mit dem Zeichen der Linker-Option, dem Schrägstrich (/). Die Groß- und Kleinschreibung spielt für Linker-Optionen keine Rolle; zum Beispiel sind /NOI und /noi identisch.

Linker-Optionen werden oft abgekürzt, um Mühe und Platz zu sparen. Die kürzeste gültige Abkürzung für jede Option wird von der Syntax der Option vorgegeben. Zum Beispiel beginnen mehrere Optionen mit den Buchstaben "NO"; daher müssen eindeutige Abkürzungen für diese Optionen länger als "NO" sein. "NO" kann nicht als Abkürzung für die Option /NOIGNORECASE verwendet werden, da der Linker zwischen den mit "NO" beginnenden Optionen nicht unterscheiden kann. Die kürzeste gültige Abkürzung für diese Option lautet /NOI.

Abkürzungen müssen mit dem ersten Buchstaben der Option beginnen und bis zum letzten eingegebenen Buchstaben zusammenhängend sein. Auslassungen und Änderungen der Buchstabenreihenfolge sind nicht zulässig.

Einige Linker-Optionen arbeiten mit numerischen Argumenten. Ein numerisches Argument kann wie folgt aussehen:

- Eine Dezimalzahl von 0 bis 65.535.
- Eine oktale Zahl von 0 bis 0177777. Eine Zahl wird als oktal interpretiert, wenn sie mit einer Null (0) beginnt. Die Zahl `10` ist zum Beispiel eine Dezimalzahl, aber die Zahl `010` ist eine Oktalzahl, die gleich dezimal 8 ist.
- Eine hexadezimale Zahl von 0 bis 0xFFFF. Eine Zahl wird als hexadezimale Zahl interpretiert, wenn sie mit einer von einem x oder X gefolgten 0 beginnt. Zum Beispiel ist `0x10` eine hexadezimale Zahl, die gleich dezimal 16 ist.

Unabhängig davon, wo die Optionen angegeben werden, wirken sich Linker-Optionen auf alle Dateien des Bindevorganges aus.

Wenn Sie beim Binden normalerweise immer dieselben Linker-Optionen verwenden, können Sie die Umgebungsvariable LINK in DOS zur Angabe bestimmter Optionen im Bindevorgang verwenden. Wenn Sie diese Variable setzen, überprüft sie der Linker auf Optionen und erwartet, daß diese wie die auf der Befehlszeile eingetippten Optionen aufgeführt sind. In der LINK-Umgebungsvariablen können keine Dateinamenargumente angegeben werden.

Hinweis Eine Befehlszeilen-Option überschreibt die Auswirkung jeder Umgebungsvariablen-Option, mit der die Befehlszeilen-Option kollidiert. Die Befehlszeilen-Option `/SE:256` zum Beispiel hebt die Auswirkung der Umgebungsvariablen-Option `/SE:512` auf.

Um zu verhindern, daß eine Option der Umgebungsvariablen verwendet wird, müssen Sie die Umgebungsvariable selbst neu setzen.

Beispiel

Im folgenden Beispiel wird die Datei *test.obj* mit den Optionen `/SE:256` und `/CO` gebunden. Anschließend wird die Datei *prog.obj* sowohl mit der Option `/NOD` als auch mit den Optionen `/SE:256` und `/CO` gebunden.

```
SET LINK=/SE:256 /CO
LINK TEST;
LINK /NOD PROG;
```

G.4.6.1 Anzeigen der Optionsliste (/HE)

/HE[LP]

Die Option /HE weist den Linker an, eine Liste der verfügbaren LINK-Optionen auf dem Bildschirm anzuzeigen.

G.4.6.2 Anhalten während des Bindens (/PAU)

/PAU[SE]

Die Option /PAU weist den Linker an, den Bindevorgang vorübergehend zu unterbrechen und vor Schreiben der ausführbaren Datei auf die Diskette eine Meldung anzuzeigen. Diese Pause ermöglicht das Einlegen einer neuen Diskette, auf die die ausführbare Datei geschrieben wird.

Bei Angabe der Option /PAUSE gibt der Linker vor Erstellung der ausführbaren Datei folgende Meldung aus:

```
.exe-Datei wird erzeugt
Diskette in Laufwerk Buchstabe wechseln; <EINGABE> drücken
```

Buchstabe entspricht dem aktuellen Laufwerk. Der Linker setzt die Verarbeitung bei Betätigung der EINGABETASTE fort.

Hinweis Entfernen Sie auf keinen Fall die Diskette, auf der die List-Datei angelegt ist, oder die für die temporäre Datei verwendete Diskette. Falls sich eine temporäre Datei auf der auszutauschenden Diskette befindet, sollte der Vorgang durch STRG+C beendet werden. Organisieren Sie die Dateien um, so daß sowohl die temporäre als auch die ausführbare Datei auf dieselbe Diskette geschrieben werden kann. Versuchen Sie anschließend erneut zu binden.

G.4.6.3 Anzeigen von Informationen über den Bindevorgang (/I)

/I[NFORMATION]

Die Option /I erteilt Informationen über den Bindevorgang, einschließlich der Phase des Bindens und der Namen der gebundenen Objektdateien.

Diese Option hilft Ihnen, die Lage und die Reihenfolge der zu bindenden Objektdateien zu bestimmen.

G.4.6.4 Unterdrücken der Linker-Anfragen (/B)

/B[ATCH]

Die Option /B teilt dem Linker mit, den Benutzer nicht nach einem neuen Pfadnamen zu fragen, wenn der Linker die benötigte Bibliothek oder Objektdatei nicht finden kann. Wird diese Option verwendet, setzt der Linker die Ausführung einfach ohne die fragliche Datei fort.

Diese Option kann ungelöste externe Bezüge verursachen. Sie ist in erster Linie für Benutzer gedacht, die Stapel- oder MAKE-Dateien zur Bindung zahlreicher ausführbarer Dateien mit einem einzigen Befehl einsetzen, und die keine Unterbrechung der Verarbeitung wünschen, falls der Linker eine benötigte Datei nicht finden kann. Diese Option ist auch bei der Umleitung der LINK-Befehlszeile behilflich, um eine Datei mit Linker-Ausgaben für zukünftige Bezugnahme zu erzeugen. Diese Option verhindert jedoch nicht, daß der Linker nach Argumenten fragt, die auf der LINK-Befehlszeile fehlen.

G.4.6.5 Erstellung von Quick-Bibliotheken (/Q)

/Q[UICKLIB]

Die Option /Q weist den Linker an, die angegebenen Objektdateien in einer Quick-Bibliothek zusammenzufassen. Beim Starten der QuickBASIC-Umgebung, können Sie die Option /L zum Laden der Quick-Bibliothek auf der QB-Befehlszeile angeben. Wenn Sie die Option /**Q** verwenden, dürfen Sie nicht vergessen, in der Liste der Bibliotheken *bqb50.lib* anzugeben, um die QuickBASIC-Unterstützungsroutinen für Quick-Bibliotheken einzugliedern.

Weitere Informationen zum Anlegen und Laden von Quick-Bibliotheken finden Sie im Anhang H, "Erstellen und Verwenden von Quick-Bibliotheken".

Hinweis Sie können die Optionen /EXEPACK und /Q nicht zusammen verwenden.

G.4.6.6 Packen der ausführbaren Dateien (/E)

/E[XEPACK]

Die Option /E entfernt Folgen sich wiederholender Bytes (typischerweise Nullzeichen) und optimiert die "Ladezeit-Verschiebungstabelle" vor Anlegen der ausführbaren Datei. Die Ladezeit-Verschiebungstabelle ist eine Referenztabelle, die sich auf den Beginn des Programmes bezieht. Der jeweilige Bezug wechselt, wenn das ausführbare Bild in den Speicher geladen und eine aktuelle Adresse für den Eingangspunkt zugewiesen wird.

Hinweis Mit dieser Option gebundene ausführbare Dateien sind kleiner und werden schneller geladen als Dateien, die ohne diese Option gebunden sind.

G.4.6.7 Ausschalten der Option Segment-Packen (/NOP)

/NOP[ACKCODE]

Die Option /NOP ist meistens nicht erforderlich, da das Code-Segment-Packen normalerweise ausgeschaltet ist. Falls eine Umgebungsvariable, wie zum Beispiel LINK, Code-Segment-Packen automatisch einschaltet, wird die Option /NOP zum Ausschalten von Segment-Packen eingesetzt.

G.4.6.8 Ignorieren der üblichen BASIC-Bibliotheken (/NOD)

/NOD[EFAULTLIBRARYSEARCH]

Beim Anlegen einer Objektdatei fügt BC die Namen der "Standard"-Bibliotheken ein - Bibliotheken, die LINK zur Auflösung externer Bezüge durchsucht. Die Option /NOD weist LINK an, *keine* in einer Objektdatei angegebene Bibliothek zu durchsuchen, um externe Bezüge aufzulösen.

Im allgemeinen laufen QuickBASIC Programme ohne die QuickBASIC-Standardbibliotheken (*brun45.lib* und *bcom45.lib*) nicht richtig. Daher sollte beim Verwenden der Option /NOD der Pfadname der erforderlichen Standardbibliothek ausdrücklich angegeben werden.

G.4.6.9 Ignorieren der Wortverzeichnisse (/NOE)

/NOE[XTDICTIONARY]

Falls LINK vermutet, daß ein globales Symbol redefiniert wurde, fordert es den Benutzer zur erneuten Bindung anhand der Option /NOE auf. Anschließend sucht es zur Lösung der Konflikte nach den einzelnen Objektdateien, nicht nach den erstellten "Wortverzeichnissen".

G.4.6.10 Setzen der Höchstanzahl an Segmenten (/SE)

/SE[GMENTS]:*Anzahl*

Die Option /SE steuert die von LINK einem Programm zugewiesene Segmentanzahl. Der Vorgabewert beträgt 128. Sie können *Anzahl* aber auf jeden beliebigen als Dezimal-, Oktal- oder Hexadezimalzahl angegebenen Wert in dem Bereich von 1 bis 1024 (dezimal) setzen.

LINK muß für jedes Segment Speicherplatz reservieren, um Segment-Informationen festzuhalten. Wenn die Segmentgrenze 128 überschreitet, weist der Linker mehr Platz für Segment-Informationen zu. Für Programme mit weniger als 128 Segmenten kann der vom Linker benötigte Speicherbetrag minimiert werden, indem man in *Anzahl* die aktuelle Segmentanzahl im Programm angibt. Falls diese Anzahl für den verfügbaren Speicherbetrag zu hoch ist, gibt LINK eine Fehlermeldung aus.

G.4.6.11 Erstellung einer Map-Datei (/M)

/M[AP]

Die Option /M erstellt eine Map-Datei. Eine Map-Datei listet die Segmente und globalen Symbole des Programms auf. LINK versucht stets, den gesamten verfügbaren Speicher zuzuweisen, um globale Symbole zu sortieren. Wenn die Symbolanzahl den Speicherplatz überschreitet, erzeugt der Linker eine unsortierte Liste. Die Datei *Mapdatei* enthält eine Liste von Symbolen, die nach Adressen sortiert ist, jedoch keine nach Namen sortierte Liste. Es folgt ein Beispiel für eine Map-Datei:

```
Start   Stop    Length Name      Class
 00000H 01E9FH 01EA0H _TEXT      CODE
 01EA0H 01EA0H 00000H C_ETEXT    ENDCODE
.
.
.
.
```

Die Informationen der mit `Start` (Beginn) und `Stop` (Ende) bezeichneten Spalten zeigen die 20-Bit-Adresse (hexadezimal) jedes Segmentes, die sich auf den Anfang des Lademoduls bezieht. Das Lademodul beginnt mit der Dateilage Null. Die mit `Length` bezeichnete Spalte gibt die Länge des Segmentes in Bytes an. Die mit `Name` bezeichnete Spalte gibt den Namen des Segmentes an und die mit `Class` bezeichnete Spalte erteilt Informationen über den Typ des Segmentes. Weitere Informationen zu Gruppen, Segmenten und Klassen finden Sie im *Microsoft MS-DOS Programmer's Reference*.

Die Startadresse sowie der Name jeder Gruppe erscheint unter der Liste der Segmente. Nachfolgend ein Beispiel für ein Gruppenlisting:

```
Origin   Group
01EA:0   DGROUP
```

In diesem Beispiel ist DGROUP der Name der Datengruppe.

Die weiter unten dargestellte Map-Datei enthält zwei Listen globaler Symbole: Die erste Liste enthält die in Reihenfolge der ASCII-Zeichen sortierten Symbolnamen; die zweite ist nach Symboladressen sortiert. Die Bezeichnung `Abs` erscheint neben den Namen absoluter Symbole (Symbole, die 16-Bit-Konstantenwerte enthalten, die nicht mit Programmadressen verknüpft sind).

Viele der in der Map-Datei erscheinenden globalen Symbole sind Symbole, die intern von Compiler und Linker verwendet werden. Sie beginnen normalerweise mit den Zeichen `B$` oder enden mit `QQ` .

```
Address           Publics by Name

01EA:0096         STKHQQ
0000:1D86         B$Shell
01EA:04B0         _edata
01EA:0910         _end
.
.
.
01EA:00EC         __abrkp
01EA:009C         __abrktb
01EA:00EC         __abrktbe
0000:9876   Abs   __acrtmsg
0000:9876   Abs   __acrtused
.
.
.
01EA:0240         ___argc
01EA:0242         ___argv

Address           Publics by Value

0000:0010         _main
0000:0047         _htoi
.
.
.
```

Die Adressen der externen Symbole liegen in dem Format *Rahmen:Offset* vor und zeigen die Lage des Symbols relativ zu Null (dem Beginn des Lademoduls) an.

Nach der Liste der Symbole gibt die Map-Datei die Startadresse des Programmes an, wie in dem folgenden Beispiel gezeigt:

```
Program entry point at 0000:0129
```

(Startadresse des Programmes bei 0000:0129)

Eine Map-Datei kann auch durch Angabe des Namens einer Map-Datei auf der LINK-Befehlszeile oder durch Eingabe des Namens einer Map-Datei als Antwort auf die Anfrage "List-Datei" festgelegt werden.

G.4.6.12 Einfügen von Zeilennummern in eine Map-Datei (/LI)

/LI[NENUMBERS]

Die Option /LI erzeugt eine Map-Datei und fügt die Zeilennummern sowie die zugeordneten Adressen des Quellprogrammes ein. Falls Sie in einzelnen Schritten kompilieren und binden, wirkt sich diese Option nur beim Binden von Objektdateien aus, die mit der Option /M kompiliert wurden.

G.4.6.13 Packen angrenzender Segmente (/PAC)

/[NO]PAC[KCODE][:*Anzahl*]

Die Option /PAC weist den Linker an, benachbarte Code-Segmente zu gruppieren. Code-Segmente derselben Gruppe benutzen gemeinsam dieselbe Segmentadresse; alle Offsetadressen werden anschließend nach Bedarf nach oben ausgerichtet. Als Ergebnis benutzen viele Befehle, die andernfalls unterschiedliche Segmentadressen haben würden, dieselbe Segmentadresse.

Falls angegeben, ist *Anzahl* das Größenlimit der von /PAC gebildeten Gruppen. LINK beendet das Hinzufügen von Segmenten an eine bestimmte Gruppe, sobald es kein Segment zu der Gruppe hinzufügen kann, ohne *Anzahl* zu überschreiten. Unter diesen Bedingungen beginnt LINK mit den verbleibenden Code-Segmenten eine neue Gruppe zu bilden. Wenn *Anzahl* nicht angegeben ist, beträgt der Vorgabewert 65.530.

Obwohl der Linker keine benachbarten Segmente packt, ohne ausdrücklich in diesem Sinn angewiesen zu werden, können Sie die Option /NOPACKCODE zum Ausschalten des Segment-Packens verwenden, wenn Sie zum Beispiel in der Umgebungsvariablen LINK die Option /PAC angegeben haben.

G.4.6.14 Der Einsatz des CodeView-Debuggers (/CO)

/CO[DEVIEW]

Die Option /CO bereitet eine ausführbare Datei für das Debuggen mit dem CodeView-Debugger vor. Falls Sie in separaten Schritten kompilieren und binden, wirkt sich diese Option nur beim Binden von Objektdateien aus, die mit der /ZI-Option des Befehls BC kompiliert sind. Auch sollte diese Option nicht mit der /Q-Option des LINK-Befehls verwendet werden, da eine Quick-Bibliothek nicht mit dem CodeView-Debugger auf Fehler untersucht werden kann.

G.4.6.15 Unterscheiden zwischen der Groß- und Kleinschreibung (/NOI)

/NOI[GNORECASE]

Die Option /NOI weist LINK an, zwischen Groß- und Kleinbuchstaben zu unterscheiden; LINK würde beispielsweise die Namen ABC, abc und Abc als drei unterschiedliche Namen erkennen. Beim Binden ist die Option /NOI nicht auf der LINK-Befehlszeile anzugeben.

G.4.7 Andere LINK-Befehlszeilen-Optionen

Nicht alle Optionen des LINK-Befehls eignen sich für die Verwendung mit QuickBASIC-Programmen. Nachstehende LINK-Optionen können zwar mit Microsoft QuickBASIC benutzt werden, sind aber überflüssig, da deren Funktionen von dem Befehl BC oder von QuickBASIC automatisch durchgeführt werden.

Option	***Funktion***
/CP[ARMAXALLOC]:*Anzahl*	Setzt die Höchstanzahl der im Programm benötigten 16-Byte-Paragraphen auf *Anzahl*, einer Ganzzahl zwischen 1 und einschließlich 65.535 fest, wenn das Programm in den Speicher geladen wird. Das Betriebssystem benutzt diesen Wert bei der Speicherplatzzuweisung vor Laden des Programms. Diese Option kann zwar auf der LINK-Befehlszeile verwendet werden, aber sie ist wirkungslos, da das BASIC-Programm den Speicher während des Programmablaufes steuert.
/DO[SSEG]	Verursacht die Anordnung der Segmente mit Hilfe der Vorgabewerte für höhere Microsoft Programmiersprachen. QuickBASIC Programme verwenden diese Segmentanordnung standardmäßig.

Option	***Funktion***
/ST[ACK]:*Anzahl*	Gibt die Stapelgröße des Programms an, wobei *Anzahl* die Stapelgröße in Bytes darstellt und jeden (dezimalen, oktalen oder hexadezimalen) positiven Wert bis 65.535 (dezimal) annehmen kann. Die BASIC Standardbibliothek setzt die vorgegebene Stapelgröße auf 2K.
/DS[ALLOCATE]	Lädt alle Daten ab dem oberen Ende des Standard-Datensegmentes.
HI[GH]	Plaziert die ausführbare Datei so hoch wie möglich im Speicher.
/NOG [ROUPASSOCIATION]	Weist den Linker an, Gruppenverbände zu ignorieren, wenn den Daten- und Codegrößen Adressen zugewiesen werden.
/O[VERLAYINTERRUPT]: *Anzahl*	Legt für die Übergabe der Kontrolle an ein Overlay eine andere Interrupt-Nummer als 0x3F fest.

Hinweis Verwenden Sie die Optionen /DS, /HI, /NOG oder /O nicht, wenn Sie mit BC kompilierte Objektdateien binden. Diese Optionen sind nur für Objektdateien geeignet, die mit dem Microsoft Macro Assembler (MASM) erstellt wurden.

G.5 Verwaltung selbständiger Bibliotheken: LIB

Der Microsoft-Bibliotheksmanager (LIB) verwaltet den Inhalt einer selbständigen Bibliothek. Eine selbständige Bibliothek besteht aus "Objektmodulen" - das heißt, aus in einer Bibliothek zusammengefaßten Objektdateien. Anders als eine Objektdatei existiert ein Objektmodul nicht unabhängig von seiner jeweiligen Bibliothek und es hat keinen Pfadnamen bzw. keine mit seinem Dateinamen verknüpfte Erweiterung. Mit LIB können Sie:

- Objektdateien kombinieren, um eine neue Bibliothek zu erstellen.
- Objektdateien in eine existierende Bibliothek einfügen.
- Objektmodule einer bestehenden Bibliothek löschen oder ersetzen.
- Objektmodule aus einer existierenden Bibliothek herausnehmen und diese in separaten Objektdateien ablegen.
- Inhalte zweier existierender Bibliotheken in einer neuen Bibliothek zusammenfassen.

Bei der Aktualisierung einer existierenden Bibliothek führt LIB sämtliche Operationen mit einer Kopie der Bibliothek durch. Diese Vorgehensweise gewährleistet eine Sicherungskopie jeder Bibliothek, falls mit der aktualisierten Version Probleme auftreten.

G.5.1 LIB starten

Der Befehl LIB kann wie folgt eingegeben werden:

- Geben Sie die Eingaben in einer Befehlszeile folgender Form an:

  LIB *alteBibl* [/P[AGESIZE]:*Nummer*] [*Befehle*][,[*List-Datei*][,[*neueBibl*]]][;]

  Die Befehlszeile darf nicht länger als 128 Zeichen sein.

- Geben Sie

  ```
  lib
  ```

  ein und beantworten Sie folgende Anfragen:

  ```
  Bibliotheksname:
  Operationen:
  List-Datei:
  Ausgabebibliothek:
  ```

  Um für eine Anfrage mehrere Dateien anzugeben, schreiben Sie ein kaufmännisches Und (&) an das Ende der Zeile. Die Anfrage erscheint erneut auf der nächsten Zeile und Sie können mit der Eingabe für die Anfrage fortfahren.

- Installieren Sie eine Datei mit Antworten auf die Anfragen des LIB-Befehls, die als "Antwortdatei" (Response File) bezeichnet wird, und geben Sie anschließend einen LIB-Befehl folgender Form ein:

  LIB @*Dateiname*

  Hierbei ist *Dateiname* der Name der Antwortdatei. Die Antworten müssen in derselben Reihenfolge stehen wie die oben erläuterten LIB-Anfragen. Außerdem können Sie den Namen einer Antwortdatei nach jeder Linker-Anfrage oder an beliebiger Stelle auf der LIB-Befehlszeile eingeben.

Tabelle G.3 zeigt die Eingaben, die Sie auf der LIB-Befehlszeile oder als Antwort auf jede Anfrage vornehmen müssen. Wenn Sie eine Antwortdatei benutzen, muß jede Antwort den Regeln dieser Tabelle folgen.

Tabelle G.3 Eingabe auf der LIB-Befehlszeile

Feld	***Anfrage***	***Eingabe***
alteBibl	"Bibliotheksname"	Name der veränderten oder angelegten Bibliothek. Wenn diese Bibliothek nicht existiert, fragt LIB, ob diese angelegt werden soll. Geben Sie den Buchstaben "y" ein, um eine neue Bibliothek anzulegen, oder den Buchstaben "n", um LIB zu beenden. Diese Meldung wird bei der Eingabe von Befehlszeichen, Kommata oder Semikolons nach dem Namen der Bibliothek unterdrückt. Ein Semikolon weist LIB an, für die Bibliothek eine Prüfung auf Übereinstimmung vorzunehmen; in diesem Fall gibt LIB eine Meldung aus, wenn es Fehler in einem Bibliotheksmodul findet.
/P:*Zahl*	/P:*Zahl* nach der Anfrage des "Bibiliotheknamens"	Die Seitengröße der Bibliothek. Diese Eingabe setzt die Seitengröße der Bibliothek auf *Zahl*, wobei *Zahl* eine ganzzahlige Potenz von 2 zwischen 16 und einschließlich 32.768 sein kann. Die Standard-Seitengröße für eine neue Bibliothek beträgt 16 Bytes. Module in dieser Bibliothek werden auf ein Vielfaches der Seitengröße (in Bytes) ab Dateianfang ausgerichtet.
/I	Keine	Weist LIB an, die Groß- und Kleinschreibung standardmäßig beim Vergleich der Symbole zu ignorieren. Dieses Feld ist bei der Zusammenfassung von Bibliotheken zu verwenden, die zwischen Groß- und Kleinschreibung unterscheiden.
/NOE	Keine	Weist LIB an, keine erweiterte Bibliothek zu erzeugen.
/NOI	Keine	Weist LIB an, beim Vergleich der Symbole auch die Groß- und Kleinschreibung zu vergleichen (LIB achtet weiterhin auf die Groß- und Kleinschreibung).
Befehle	"Operationen "	Befehlssymbole sowie Objektdateien, die LIB die in der Bibliothek vorzunehmenden Änderungen mitteilen.

Fortsetzung auf der folgenden Seite.

Feld	***Anfrage***	***Eingabe***
List-Datei	"List-Datei"	Name einer List-Datei für Querverweise. Es wird keine List-Datei erzeugt, wenn kein Dateiname angegeben wird.
neuBibl	"Ausgabebibliothek"	Name der veränderten Bibliothek, die LIB als Ausgabe erzeugt. Falls Sie keinen neuen Dateinamen angeben, wird die ursprüngliche, unveränderte Datei in einer Bibliotheksdatei mit demselben Namen, aber der Erweiterung *.bak*, die die Erweiterung *.lib* ersetzt, gespeichert.

G.5.2 Übliche Vorgaben für LIB

LIB hat eigene eingebaute, als Vorgaben bezeichnete Antworten. Sie können diese Vorgaben für jede von LIB benötigte Information auf eine der folgenden Arten wählen:

- Um für einen Eintrag der Befehlszeile die Vorgabe zu wählen, lassen Sie den Dateinamen oder die Dateinamen vor dem Eintrag aus und geben Sie nur das erforderliche Komma ein. Die einzige Ausnahme zu dieser Regel ist die Vorgabe für den Eintrag der *List-Datei*: Wird ein Komma als Platzhalter für diesen Eintrag verwendet, erzeugt LIB eine List-Datei mit Querverweisen.
- Um die Vorgabe für jede Anfrage zu wählen, ist einfach die EINGABETASTE zu betätigen.
- Um die Vorgabewerte für alle Einträge oder Anfragen der Befehlszeile, die einem Eintrag oder einer Anfrage folgen, zu wählen, geben Sie ein Semikolon (;) nach dem Eintrag oder der Anfrage ein. Das Semikolon sollte das letzte Zeichen auf der Befehlszeile sein.

Die folgende Liste zeigt die Vorgabewerte, die LIB für List-Dateien mit Querverweisen und Ausgabebibliotheken verwendet.

Datei	***Vorgabe***
Liste mit Querverweisen	Der spezielle Dateiname NUL, der dem Linker mitteilt, *keine* List-Datei mit Querverweisen anzulegen.
Ausgabebibliothek	Der Eintrag *alteBibl* oder die Antwort auf die Anfrage "Bibliotheksname".

G.5.3 List-Dateien mit Querverweisen

Eine List-Datei mit Querverweisen stellt fest, welche Routinen sich in einer selbständigen Bibliothek befinden und aus welchen ursprünglichen Objektdateien diese stammen. Eine List-Datei mit Querverweisen enthält folgende Listen:

- Eine alphabetische Liste aller globalen Symbole der Bibliothek. Jedem Symbolnamen folgt der Name des Moduls, in dem das Symbol definiert wird.
- Eine Liste der Module in der Bibliothek. Unter jedem Modulnamen befindet sich eine alphabetische Liste der in diesem Modul definierten globalen Symbole.

G.5.4 Befehlssymbole

Um LIB die in der Bibliothek durchzuführenden Änderungen mitzuteilen, geben Sie ein Befehlssymbol, wie zum Beispiel +, -, - +, * oder -*, unmittelbar gefolgt von einem Modulnamen, Namen einer Objektdatei oder Namen einer Bibliothek ein. Sie können mehr als eine Operation in beliebiger Reihenfolge angeben.

Die folgende Liste zeigt jedes LIB-Befehlssymbol, den Typ des mit dem Symbol anzugebenden Dateinamens, und die Funktion des Symbols an:

Befehl	***Bedeutung***
+{*Objdatei* | *Bibl*}	Fügt die angegebene Objektdatei der eingegebenen Bibliothek hinzu und läßt diese Objektdatei das letzte Modul in der Bibliothek werden, falls dem Befehl der Name einer Objektdatei angegeben wurde. Sie können einen Pfadnamen für den Namen der Objektdatei verwenden. Da LIB automatisch die Erweiterung *.obj* hinzufügt, kann die Erweiterung für den Namen der Objektdatei ausgelassen werden. Falls mit einem Bibliotheknamen angegeben, fügt das Pluszeichen (+) den Inhalt dieser Bibliothek in die angegebene Bibliothek ein. Der Name der Bibliothek muß die Erweiterung *.lib* aufweisen.
-*Modul*	Löscht das angegebene Modul aus der eingegebenen Bibliothek. Ein Modulname hat keinen Pfadnamen bzw. keine Erweiterung.
-+*Modul*	Ersetzt das angegebene Modul in der eingegebenen Bibliothek. Modulnamen haben keinen Pfadnamen bzw. keine Erweiterungen. *lib* löscht das angegebene Modul und fügt anschließend die Objektdatei an, die denselben Namen wie das Modul hat. Von der Objektdatei wird angenommen, daß sie die Erweiterung *.obj* hat und sich in dem aktuellen Arbeitsverzeichnis befindet.

Befehl	***Bedeutung***
**Modul*	Kopiert das angegebene Modul aus der Bibliothek in eine Objektdatei, die sich im aktuellen Arbeitsverzeichnis befindet. Das Modul verbleibt in der Bibliotheksdatei. Wenn LIB das Modul in eine Objektdatei kopiert, fügt es die Erweiterung *.obj* an. Sie können weder die Erweiterung *.obj*, noch das Ziellaufwerk oder den der Objektdatei zugewiesenen Pfadnamen überschreiben. Später jedoch können Sie die Datei umbenennen oder an jede beliebige Stelle kopieren.
-**Modul*	Bewegt das angegebene Objektmodul aus der Bibliothek in eine Objektdatei. Diese Operation ist identisch mit dem oben beschriebenen Kopieren des Moduls in eine Objektdatei und anschließendem Löschen des Moduls aus der Bibliothek.

Beispiele

Das folgende Beispiel verwendet das Befehlssymbol zum Ersetzen (- +), um LIB anzuweisen, das Modul `HEAP` in der Bibliothek *lang.lib* zu ersetzen. Das Programm LIB löscht das Modul `HEAP` aus der Bibliothek und fügt anschließend die Objektdatei *heap.obj* als neues Modul in die Bibliothek ein. Das Semikolon am Ende der Befehlszeile zeigt LIB an, daß es die Vorgaben für die verbleibenden Anfragen verwenden soll. Das bedeutet, daß keine List-Datei erzeugt wird und daß die Änderungen in die ursprüngliche Bibliotheksdatei geschrieben werden, anstatt eine neue Bibliothek zu erzeugen.

```
LIB LANG-+HEAP;
```

Die folgenden Beispiele führen dieselbe Funktion wie das erste Beispiel in diesem Abschnitt aus, allerdings in zwei separaten Operationen, die die Befehlssymbole Hinzufügen (+) und Löschen (-) verwenden. Die Ergebnisse dieser Beispiele sind identisch, da Löschoperationen immer vor Ergänzungsoperationen zum Hinzufügen ausgeführt werden, unabhängig von der Reihenfolge dieser Operationen auf der Befehlszeile. Diese Reihenfolge der Ausführung verhindert Konflikte, wenn die neue Version eines Modules eine alte Version in der Bibliotheksdatei ersetzt.

```
LIB LANG-HEAP+HEAP;
LIB LANG+HEAP-HEAP;
```

Das folgende Beispiel veranlaßt LIB, eine Prüfung auf Übereinstimmung mit der Bibliotheksdatei *for.lib* vorzunehmen. Es wird keine andere Aktion durchgeführt. LIB zeigt jeden gefundenen Übereinstimmungsfehler an und kehrt auf Betriebssystemebene zurück.

```
LIB FOR;
```

Das folgende Beispiel teilt LIB mit, eine Prüfung auf Übereinstimmung mit der Bibliotheksdatei *lang.lib* durchzuführen und anschließend eine List-Datei mit Querverweisen zu erzeugen, die *lquer.pub* heißt.

```
LIB LANG,LQUER.PUB
```

Das nächste Beispiel weist LIB an, das Modul `PUMPEN` aus der Bibliothek *erste.lib* in eine *pumpen.obj* genannte Objektdatei zu bewegen. Das Modul `PUMPEN` wird bei diesem Vorgang aus der Bibliothek entfernt. Das Modul `MEHR` wird aus der Bibliothek in eine Objektdatei kopiert, die *mehr.obj* heißt; das Modul verbleibt in der Bibliothek. Die geänderte Bibliothek heißt *zweite.lib*. Sie enthält alle Module der Bibliothek *erste.lib*, mit Ausnahme des Moduls `PUMPEN`, das durch Verwendung des Befehlssymbols Bewegen (`-*`) entfernt wurde. Die ursprüngliche Bibliothek *erste.lib* bleibt unverändert.

```
LIB ERSTE -*PUMPEN *MEHR, ,ZWEITE
```

Der Inhalt der folgenden Antwortdatei veranlaßt LIB, das Modul `HEAP` aus der Bibliotheksdatei *libfor.lib* zu löschen, das Modul `SCHWACH` herauszunehmen (ohne es zu löschen) und es in einer *schwach.obj* genannten Objektdatei abzulegen, sowie die Objektdateien *cursor.obj* und *heap.obj* als die letzten beiden Module an die Bibliothek anzuhängen. Zum Schluß erzeugt LIB eine List-Datei mit Querverweisen, die *querlst* heißt.

```
LIBFOR
+CURSOR+HEAP-HEAP*SCHWACH
QUERLST
```

G.5.5 LIB-Optionen

LIB verfügt über vier Optionen, die auf der Befehlszeile nach dem erforderlichen Dateinamen der Bibliothek und vor jeglichem Befehl einzugeben sind.

G.5.5.1 Ignorieren der Groß- und Kleinschreibung bei Symbolen

/I[GNORECASE]

Die Option /I teilt LIB mit, die Groß- und Kleinschreibung bei dem von LIB standardmäßig vorgenommenen Vergleich von Symbolen zu ignorieren. Diese Option ist bei der Zusammenfassung einer mit /NOI markierten Bibliothek (wie nachstehend beschrieben) mit anderen nicht markierten Bibliotheken zu verwenden, um eine neue, nicht markierte Bibliothek zu erstellen.

G.5.5.2 Ignorieren der erweiterten Wortverzeichnisse

/NOE[XTDICTIONARY]

Die Option /NOE weist LIB an, kein erweitertes Wortverzeichnis zu erzeugen. Das erweiterte Wortverzeichnis bildet einen getrennten Teil der Bibliothek, der dem Linker bei einer schnelleren Verarbeitung der Bibliotheken hilft.

Benutzen Sie die Option /NOE beim Auftreten der Fehler `U1171` oder `U1172` oder wenn das erweiterte Wortverzeichnis bei LINK Probleme verursacht. Nähere Informationen über die Beziehung zwischen LINK und dem erweiterten Wortverzeichnis entnehmen Sie bitte Abschnitt G.4.6.9.

G.5.5.3 Unterscheiden der Symbolschreibweise

/NOI[GNORECASE]

Die Option /NOI weist LIB an, die Groß- und Kleinschreibung bei dem Vergleich von Symbolen nicht zu ignorieren, das bedeutet, daß /NOI LIB zum Unterscheiden der Groß- und Kleinschreibung veranlaßt. Standardmäßig ignoriert LIB die Groß- und Kleinschreibung. Mit Hilfe dieser Option können lediglich durch die Schreibweise unterschiedene Symbole, wie zum Beispiel SPLINE und Spline, in derselben Bibliothek eingesetzt werden.

Bei Erstellung einer Bibliothek mit der Option /NOI. markiert LIB die Bibliothek intern mit dieser Angabe. Frühere LIB-Versionen haben Bibliotheken nicht auf diese Weise markiert. Bei der Zusammenfassung mehrerer Bibliotheken, von denen eine mit /NOI markiert ist, wird die gesamte Ausgabebibliothek mit /NOI markiert.

G.5.5.4 Festlegen der Größe einer Bibliotheksseite

/P[AGESIZE]:*Anzahl*

Die Seitengröße einer Bibliothek wirkt sich auf die Ausrichtung der in der Bibliothek gespeicherten Module aus. Module in der Bibliothek werden so ausgerichtet, daß sie auf einem Vielfachen der Seitengröße (in Bytes) ab Dateianfang beginnen. Die vorgegebene Seitengröße für eine neu angelegte Bibliothek beträgt 16 Bytes.

Sie können eine andere Bibliothek-Seitengröße setzen, während Sie eine Bibliothek anlegen oder die Seitengröße einer existierenden Bibliothek ändern, indem Sie die folgende Option nach dem *altBibl*-Eintrag auf der LIB-Befehlszeile oder nach dem Namen, den Sie als Antwort auf die Anfrage "Bibliotheksname" eingeben, einfügen:

/P[AGESIZE]:*Zahl*

Zahl legt die neue Bibliothek-Seitengröße fest. Sie muß ein ganzzahliger Wert sein, der eine Potenz von 2 zwischen den Werten 16 und 32.768 darstellt.

Die Bibliothek-Seitengröße bestimmt die Modulanzahl, die die Bibliothek enthalten kann; daher ermöglicht die Vergrößerung der Seitengröße das Einfügen mehrerer Module in die Bibliothek. Je größer die Seitengröße, desto größer ist jedoch der ungenutzte Speicherplatz in der Bibliothek (im Durchschnitt *Seitengröße*/2 Bytes). In den meisten Fällen ist eine kleine Seitengröße zu verwenden, falls nicht eine sehr große Modulanzahl in einer Bibliothek abgelegt werden soll.

Außerdem bestimmt die Seitengröße die maximal mögliche Größe der Bibliothek. Diese Obergrenze ist *Zahl* * 65.536. Wenn Sie zum Beispiel LIB mit der Option /P:16 aufrufen, muß die Bibliothek kleiner als ein Megabyte (16 * 65.536 Bytes) sein.

Anhang H: Erstellen und Verwenden von Quick-Bibliotheken

Dieser Anhang beschreibt Anlegen und Wartung von Bibliotheken aus der QuickBASIC-Programmierumgebung. Eine Bibliothek ist eine Datei, die mehrere Module unter einem einzigen Dateinamen beinhaltet. In diesem Anhang lernen Sie folgende Verfahren:

- Erstellen von Bibliotheken aus der QuickBASIC-Umgebung.
- Laden einer Quick-Bibliothek, während eines QuickBASIC-Programmablaufes.
- Ansehen des Inhaltes einer Quick-Bibliothek.
- Routinen, die in anderen Sprachen geschrieben sind, in eine Quick-Bibliothek einzufügen.

H.1 Bibliotheksarten

QuickBASIC bietet Dienstprogramme zur Erstellung zweier verschiedener Bibliotheksarten, die durch unterschiedliche Erweiterungen der Dateinamen gekennzeichnet sind:

Erweiterung	***Funktion***
.qlb	Die Erweiterung *.qlb* kennzeichnet eine Quick-Bibliothek, eine besondere Art von Bibliothek, die eine einfache Ergänzung verschiedener Programme mit häufig verwendeten Prozeduren erlaubt. Eine Quick-Bibliothek kann Prozeduren enthalten, die in QuickBASIC oder anderen Microsoft-Sprachen, wie zum Beispiel Microsoft® C, geschrieben sind.
.lib	Die Erweiterung *.lib* kennzeichnet eine selbständige mit dem Microsoft Library Manager, LIB, erstellte (*.lib*) Bibliothek. Wenn QuickBASIC eine Quick-Bibliothek anlegt, legt es gleichzeitig eine *.lib*-Bibliothek an, die dieselben Prozeduren in leicht abgewandelter Form enthält.

Beide Bibliotheksarten können sowohl aus der Programmierumgebung als auch aus der Befehlszeile angelegt werden. Sie können sich eine Quick-Bibliothek als eine Gruppe von Prozeduren vorstellen, die beim Laden der Bibliothek in QuickBASIC "eingegliedert" werden. Bibliotheken mit der Erweiterung *.lib* sind im wesentlichen selbständige, kompilierte Prozeduren. Sie können entweder einer Quick-Bibliothek hinzugefügt oder mit einem Hauptmodul gebunden werden, um eine aus der DOS-Befehlszeile ausführbare Datei zu erzeugen.

Dieser Anhang erläutert die Verwendung von Befehlszeilen-Dienstprogrammen für allgemeine Fälle. Eine umfassende Erläuterung zur Verwendung solcher Hilfsprogramme entnehmen Sie bitte Anhang G, "Kompilieren und Binden aus DOS".

H.2 Vorteile der Quick-Bibliotheken

Quick-Bibliotheken erleichtern die Programmentwicklung und -pflege. Während der Entwicklung eines Projektes werden Module fester Bestandteil des Programmes und können in eine Quick-Bibliothek eingefügt werden. Anschließend können Sie die Quelldateien der Originalmodule beiseite legen bis Sie sich entschließen, sie zu verbessern oder beizubehalten. Die Bibliothek kann jetzt gleichzeitig mit QuickBASIC geladen werden und das Programm hat sofortigen Zugriff auf alle Prozeduren der Bibliothek.

Prozeduren einer Quick-Bibliothek verhalten sich wie QuickBASIC-Anweisungen. Eine korrekt deklarierte **SUB**-Prozedur der Quick-Bibliothek kann sogar ohne eine **CALL**-Anweisung aufgerufen werden. Weitere Informationen zum Aufruf einer **SUB**-Prozedur mit oder ohne dem Schlüsselwort **CALL** finden Sie im Kapitel 2, "Prozeduren: Unterprogramme und Funktionen".

Prozeduren einer Quick-Bibliothek können genau wie BASIC-Anweisungen unmittelbar aus dem Direkt-Fenster ausgeführt werden. Das bedeutet, daß deren Auswirkungen vor dem Einsatz in anderen Programmen getestet werden können.

Wenn Sie zusammen mit anderen Programmierern Programme entwickeln, erleichtern Quick-Bibliotheken die Aktualisierung gemeinsam benutzter Prozeduren. Wenn Sie eine Bibliothek mit Originalprozeduren zur kommerziellen Nutzung anbieten, sind ebenfalls alle QuickBASIC-Programmierer in der Lage, diese Prozeduren sofort für eigene Anwendungen einzusetzen. Sie können die selbsterstellte Quick-Bibliothek potentiellen Kunden vor dem Kauf für Demozwecke zur Verfügung stellen. Da Quick-Bibliotheken keinen Quellcode enthalten und nur innerhalb der QuickBASIC-Programmierumgebung verwendbar sind, werden Ihre Urheberrechte geschützt, während Ihre Marktchancen verbessert werden.

Hinweis Quick-Bibliotheken haben dieselbe Funktion wie Benutzer-Bibliotheken in QuickBASIC Version 2.0 bzw. 3.0. Eine Benutzer-Bibliothek läßt sich jedoch nicht als Quick-Bibliothek laden. Die Bibliothek muß mit dem ursprünglichen Quellcode, wie unten beschrieben, neu angelegt werden.

H.3 Anlegen einer Quick-Bibliothek

Eine Quick-Bibliothek enthält automatisch sämtliche Module, sowohl das Hauptmodul als auch alle weiteren Module, die sich beim Anlegen der Bibliothek innerhalb der QuickBASIC-Umgebung befinden. Darüber hinaus beinhaltet sie die jeweilige, beim Start von QuickBASIC geladene Quick-Bibliothek. Falls Sie beim Laden eines Programms nur bestimmte Module in der Bibliothek ablegen möchten, müssen Sie die unerwünschten Module ausdrücklich entfernen. Benutzen Sie dazu den Befehl **Datei entfernen** aus dem Menü **Datei**.

Das Verzeichnisfeld des Befehls **SUBs** aus dem Menü **Ansicht** gewährt einen schnellen Überblick der geladenen Module. Diese Methode zeigt die Prozeduren einer geladenen Bibliothek jedoch nicht an. Das in Abschnitt H.4.3, "Anzeigen des Inhalts einer Quick-Bibliothek", dargestellte Hilfsprogramm *qbibdru.bas*, listet alle Prozeduren einer Bibliothek auf.

In eine Bibliothek können nur ganze Module geschrieben werden, das bedeutet, daß keine einzelne Prozeduren eines Moduls ausgewählt werden können. Wenn Sie nur bestimmte Prozeduren eines Moduls einbinden möchten, schreiben Sie die gewünschten Prozeduren in ein separates Modul und legen Sie dieses Modul anschließend in einer Bibliothek ab.

Eine Quick-Bibliothek muß abgeschlossen sein, das heißt, sie darf nur Prozeduren derselben Bibliothek aufrufen. Auch müssen Prozedurnamen innerhalb der Bibliothek eindeutig sein.

Bei großen Programmen können Sie die Ladezeit reduzieren, indem Sie so viele Routinen wie möglich in Quick-Bibliotheken ablegen. Das Ablegen zahlreicher Routinen in Quick-Bibliotheken stellt darüber hinaus einen Vorteil dar, wenn das Programm später in eine selbständig ausführbare Datei überführt werden soll, da der Bibliothekinhalt nur gebunden, nicht neu kompiliert wird.

Hinweis Ihr Hauptmodul kann, aber muß nicht Prozeduren enthalten. Wenn es Prozeduren enthält und diese in die Bibliothek eingebunden werden, wird auch das Hauptmodul selbst in der Bibliothek abgelegt. Dieser Vorgang erzeugt zwar keine Fehlermeldung, aber der Modul-Ebenen-Code der Bibliothek kann nicht ausgeführt werden, wenn deren Prozeduren keine Routine (wie zum Beispiel **ON ERROR**) enthalten, die die Kontrolle ausdrücklich an die Modul-Ebene übergibt. Selbst in diesem Fall, könnte der Großteil des Modul-Ebenen-Codes unwesentlich sein. Wenn Sie Prozeduren in häufig zusammen benutzten Modulen organisieren, werden die Quick-Bibliotheken wahrscheinlich weniger überflüssigen Code enthalten.

H.3.1 Zum Anlegen einer Quick-Bibliothek erforderliche Dateien

Vergewissern Sie sich vor dem Anlegen einer Quick-Bibliothek, daß Ihnen die richtigen Dateien zur Verfügung stehen. Wenn Sie nicht über eine Festplatte verfügen, sollten die Dateien und weiteren Programme auf verschiedenen Disketten gespeichert werden. QuickBASIC fragt nach einem Pfadnamen, wenn es eine Datei nicht finden kann. In diesem Fall legen Sie die richtige Diskette ein und beantworten Sie die Anfrage.

Stellen Sie sicher, daß sich die folgenden Dateien im aktuellen Arbeitsverzeichnis befinden oder für QuickBASIC durch die entsprechenden DOS-Umgebungsvariablen verfügbar sind:

Datei	***Zweck***
qb.exe	Steuert den Erstellungsprozeß einer Quick-Bibliothek. Falls Sie nur mit QuickBASIC-Modulen arbeiten, können Sie den gesamten Vorgang in einem Schritt innerhalb der QuickBASIC-Umgebung ausführen.
bc.exe	Erstellt Objektdateien aus dem Quellcode.
link.exe	Bindet Objektdateien.
lib.exe	Verwaltet selbständige Bibliotheken von Objektmodulen.
bqb50.lib	Stellt die von der Quick-Bibliothek benötigten Routinen zur Verfügung. Diese Bibliothek ist eine selbständige Bibliothek, die mit Objekten Ihrer eigenen Bibliothek gebunden wird, um eine Quick-Bibliothek zu erstellen.

H.3.2 Aufbau einer Quick-Bibliothek

Die meisten Quick-Bibliotheken werden innerhalb der QuickBASIC-Umgebung angelegt. Es kann gelegentlich vorkommen, daß Sie eine Bibliothek aktualisieren oder Routinen aus anderen Microsoft-Sprachen in die Bibliothek einbinden möchten. In diesem Fall ist die Erstellung einer Basis-Bibliothek aus den Nicht-BASIC-Routinen außerhalb der Umgebung mit dem direkten Aufruf von LINK und LIB zu beginnen. Anschließend können die aktuellsten QuickBASIC-Module innerhalb von QuickBASIC hinzugefügt werden.

H.3.3 Aufbau einer Quick-Bibliothek innerhalb der QuickBASIC-Umgebung

Beim Aufbau einer Bibliothek innerhalb der QuickBASIC-Umgebung ist zu berücksichtigen, ob es sich um eine neue Bibliothek oder um die Aktualisierung einer bereits existierenden Bibliothek handelt. Im Fall einer Aktualisierung, sollte QuickBASIC mit der Befehlszeilen-Option /L gestartet und der Name der zu aktualisierenden Bibliothek als Befehlszeilen-Argument angegeben werden. Gleichzeitig können Sie auch den Namen eines Programmes angeben, dessen Module Sie in der Bibliothek ablegen möchten. In diesem Fall lädt QuickBASIC alle in der *.mak*-Datei dieses Programms angegebenen Module.

H.3.3.1 Entfernen unerwünschter Dateien

Falls das Programm beim Start von QuickBASIC geladen wird, stellen Sie sicher, daß alle in der QuickBASIC-Bibliothek unerwünschten Module einschließlich des Hauptmoduls (es sei denn, es enthält Prozeduren, die Sie in der Bibliothek ablegen möchten) entfernt werden.

Die Module lassen sich wie folgt entfernen:

1. Wählen Sie den Befehl **Datei entfernen** aus dem Menü **Datei**.
2. Markieren Sie das zu entfernende Modul im Listenfeld und betätigen Sie anschließend die EINGABETASTE.
3. Wiederholen Sie die Schritte 1 bis 2, bis alle unerwünschten Module entfernt sind.

H.3.3.2 Laden gewünschter Dateien

Es ist ebenfalls möglich, QuickBASIC mit oder ohne eine Bibliotheksangabe einfach zu starten und die gewünschten Module aus der Umgebung nacheinander zu laden. In diesem Fall wird jedes Modul mit dem Befehl **Datei laden** aus dem Menü **Datei** geladen.

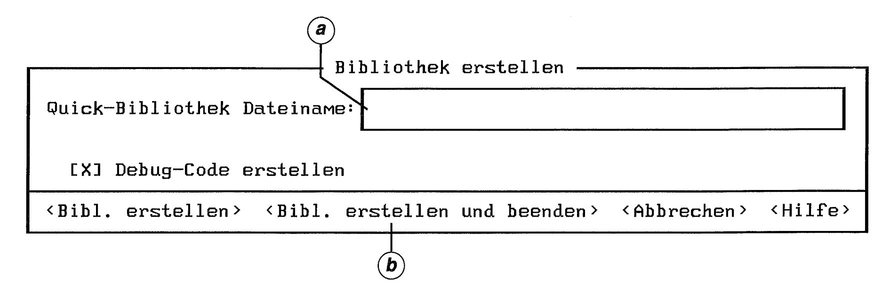
Um Module einzeln mit QuickBASIC zu laden, verfahren Sie wie folgt:

1. Wählen Sie den Befehl **Datei laden** aus dem Menü **Datei**.
2. Markieren Sie den Namen des zu ladenden Moduls im Listenfeld.
3. Wiederholen Sie die Schritte 1 bis 2, bis alle gewünschten Module geladen sind.

H.3.3.3 Erstellen einer Quick-Bibliothek

Nachdem Sie die vorhergehende Bibliothek (sofern vorhanden) sowie alle neuen Module, die Sie in die Quick-Bibliothek einbinden möchten, geladen haben, wählen Sie den Befehl **Bibliothek erstellen** aus dem Menü **Ausführen**. Es erscheint das in Abbildung H.1 gezeigte Dialogfeld.

Abbildung H.1 Erstellen des Dialogfeldes Bibliothek



a) Geben Sie den Namen der Quick-Bibliothek in diesem Feld ein.

b) Erstellt eine Bibliothek und kehrt zu DOS zurück.

Eine Quick-Bibliothek ist wie folgt zu erstellen:

1. Geben Sie den Namen der anzulegenden Bibliothek in das Textfeld **Quick-Bibliothek Dateiname** ein.

   Falls Sie nur einen Basisnamen angeben (einen Dateinamen ohne Erweiterung), fügt QuickBASIC beim Erstellen der Bibliothek die Erweiterung *.qlb* automatisch hinzu. Ist eine Erweiterung für die Bibliothek unerwünscht, muß der Basisname mit einem abschließenden Punkt (.) ergänzt werden. Im übrigen können Sie jeden beliebigen Basisnamen und jede beliebige Erweiterung eingeben, die den Regeln für DOS-Dateinamen entsprechen.

2. Markieren Sie das Kontrollfeld **Debug-Code erstellen** nur, wenn Sie einen in der zu aktualisierenden Bibliothek vermuteten Fehler verfolgen. Dies vergrößert die Bibliothek, verlangsamt den Programmablauf und deren Fehlerkontrolle beschränkt sich hauptsächlich auf die Überprüfung der Datenfeldgrenzen.

3. Legen Sie die Quick-Bibliothek an:

   - Wählen Sie die Befehlsfläche **Bibl.erstellen**, wenn Sie nach Erstellung der Quick-Bibliothek in der Umgebung bleiben möchten.

   - Wählen Sie die Befehlsfläche **Bibl.erstellen und beenden**, wenn Sie nach Erstellung der Quick-Bibliothek auf die DOS-Befehlsebene zurückkehren möchten.

Hinweis Beim Erstellen einer Quick-Bibliothek, ist zu beachten, daß mindestens eines der Module der Bibliothek eine Anweisung zur Ereignisverfolgung enthalten muß für den Fall, daß die Quick-Bibliothek mit einem Nicht-Bibliotheksmodul verwendet wird, das Ereignisverfolgung, wie zum Beispiel Tastenverfolgung, benötigt. Es kann sich um eine einfache Anweisung wie **TIMER OFF** handeln, ohne die Ereignisse in der Quick-Bibliothek jedoch nicht korrekt verfolgt werden.

H.4 Verwendung der Quick-Bibliotheken

Dieser Abschnitt erläutert das Laden der Quick-Bibliotheken, wenn Sie QuickBASIC starten und wie Sie sich den Inhalt einer Quick-Bibliothek ansehen. Außerdem erteilt er wichtige Informationen über Prozeduren, die Gleitkomma-Arithmetik innerhalb einer Quick-Bibliothek durchführen.

H.4.1 Laden einer Quick-Bibliothek

Um eine Quick-Bibliothek zu laden, müssen Sie beim Starten von QuickBASIC den Namen der gewünschten Bibliothek mit der folgenden Syntax auf der Befehlszeile angeben:

QB [*Progname*] /L [*qBiblname*]

Falls Sie QuickBASIC mit der Option /L laden und den Namen einer Bibliothek (*qBiblname* in diesem Beispiel) angeben, lädt QuickBASIC die angegebene Datei und bringt Sie in die Programmierumgebung. Der Inhalt der Bibliothek ist jetzt einsatzbereit. Falls Sie QuickBASIC mit der Option /L starten, aber keine Bibliothek angeben, lädt QuickBASIC die Bibliothek *qb.qlb* (siehe Abschnitt H.5).

Sie können QuickBASIC auch mit der Option /RUN, gefolgt von einem Programmnamen (*Progname*) und der Option /L starten. In diesem Fall lädt QuickBASIC sowohl das Programm als auch die angegebene Quick-Bibliothek und startet das Programm anschließend, ohne in der Programmierumgebung anzuhalten.

Hinweis Wenn Sie Programmodule durch Quick-Bibliotheken darstellen, ist die *.mak*-Datei zu aktualisieren, so daß sie laufend zu den Modulen des sich entwickelnden Programms angepaßt bleibt. (Dieses geschieht mit dem Befehl **Datei entfernen** aus dem Menü **Datei**.) Eine nicht aktualisierte *.mak* Datei könnte QuickBASIC veranlassen, ein Modul zu laden, das eine Prozedurdefinition mit demselben Namen wie der eines in der Quick-Bibliothek definierten Moduls hat. Diese Operation würde zu der Fehlermeldung `Doppelte Definition` führen.

Sie können nur jeweils eine Quick-Bibliothek laden. Bei Angabe eines Pfades sucht QuickBASIC an der angegebenen Stelle. Andernfalls sucht QuickBASIC nach der Quick-Bibliothek an den folgenden drei Stellen:

1. In dem aktuellen Verzeichnis.
2. Im Pfad, der vom Befehl **Suchpfade festlegen** für Bibliotheken angegeben wurde.
3. In dem Pfad, der von der LIB-Umgebungsvariablen angegeben ist. (Informationen zu Umgebungsvariablen finden Sie in der DOS-Dokumentation.)

Beispiel

Der folgende Befehl startet QuickBASIC und führt das *report.bas* genannte Programm mit den Routinen aus der Bibliothek *figs.qlb* aus:

```
QB /RUN report.bas /L figs.qlb
```

H.4.2 Gleitkomma-Arithmetik in Quick-Bibliotheken

BASIC-Prozeduren innerhalb von Quick-Bibliotheken stellen mit dem BC-Befehlszeilen-Compiler kompilierten Code dar. Diese Prozeduren haben wichtige Eigenschaften mit ausführbaren Dateien gemeinsam. Zum Beispiel führen sowohl ausführbare Dateien als auch Quick-Bibliotheken Gleitkomma-Arithmetik schneller und mit größerer Genauigkeit durch, als innerhalb der QuickBASIC-Umgebung durchgeführte Berechnungen. Nähere Informationen finden Sie in Kapitel 6, "Das Menü Ausführen", des Handbuches *Lernen und Anwenden von Microsoft QuickBASIC*.

H.4.3 Anzeigen des Inhalts einer Quick-Bibliothek

Da eine Quick-Bibliothek im wesentlichen eine Binärdatei ist, können Sie deren Inhalt nicht einfach mit einem Texteditor betrachten. Das auf den Originaldisketten vorhandene Dienstprogramm *qbibdru.bas* gewährleistet die Auflistung aller Prozeduren und Datensymbole einer bestimmten Bibliothek. Sehen Sie sich den Inhalt einer Quick-Bibliothek wie folgt an:

1. Starten Sie QuickBASIC.

2. Laden und starten Sie *qbibdru.bas*.
3. Beantworten Sie die Anfrage mit dem Namen der zu untersuchenden Quick-Bibliothek. Sie brauchen die Erweiterung *.qlb* bei Eingabe des Dateinamens nicht anzugeben, aber diese zusätzliche Angabe kann auch nicht schaden.

   Falls die angegebene Datei als Quick-Bibliothek existiert, zeigt das Programm eine Liste aller in der Bibliothek verwendeten Symbolnamen an. In diesem Zusammenhang beziehen sich Symbolnamen auf die Prozedurnamen der Bibliothek.

Ein kommentiertes Listing von *qbibdru.bas* finden Sie in Kapitel 3, "Datei- und Geräte-E/A".

H.5 Die mitgelieferte Bibliothek (*qb.qlb*)

Wenn Sie QuickBASIC mit der Option /L aufrufen ohne den Namen einer Quick-Bibliothek anzugeben, lädt QuickBASIC automatisch die im QuickBASIC Paket mitgelieferte Datei *qb.qlb* Sie enthält drei Routinen, INTERRUPT, INT86OLD und ABSOLUTE, die die Unterstützung von Software-Unterbrechungen (Interrupt) für Systemaufrufe sowie für **CALL ABSOLUTE** bieten. Um die Routinen aus *qb.qlb* zu verwenden, müssen Sie diese (oder eine andere Bibliothek, in die solche Routinen eingebunden sind) beim Aufruf von QuickBASIC auf der Befehlszeile angeben. Falls Sie diese Routinen zusammen mit anderen in Bibliotheken abgelegten Routinen laden möchten, erstellen Sie eine Kopie der Bibliothek *qb.qlb* als Basis zur Erstellung einer Bibliothek, die alle benötigten Routinen enthält.

H.6 Die Dateinamenerweiterung *.qlb*

Die Erweiterung *.qlb* ist nur eine bequeme Vereinbarung. Es kann jede beliebige Erweiterung oder auch keine Erweiterung für die Quick-Bibliotheksdateien verwendet werden. Bei der Verarbeitung der Option /L *qBiblname* geht QuickBASIC jedoch davon aus, daß der aufgeführte Name *qBiblname* die Erweiterung *.qlb* hat, wenn keine andere Erweiterung angegeben ist. Falls die jeweilige Quick-Bibliothek keine Erweiterung hat, ist ein Punkt nach deren Namen (*qBiblname.*) anzugeben; andernfalls sucht QuickBASIC nach einer Datei mit dem angegebenen Basisnamen und der Erweiterung *.qlb*.

H.7 Erstellen einer Bibliothek aus der Befehlszeile

Nach Aufbau einer Bibliothek innerhalb der QuickBASIC-Umgebung sind zusätzliche Dateien mit den Erweiterungen *.obj* und *.lib* vorhanden. Beim Anlegen von Quick-Bibliotheken steuert QuickBASIC die Arbeit dreier anderer Programme, BC, LINK und LIB, und faßt anschließend das Ergebnis sowohl in einer Quick-Bibliothek als auch in einer selbständigen (*.lib*) Bibliothek zusammen. Nach Abschluß dieses Prozesses gibt es eine Objekt-Datei (*.obj*) für jedes Modul der Quick-Bibliothek und eine einzelne Bibliotheksdatei (*.lib*), die für jede Objektdatei ein Objektmodul enthält. Die Dateien mit der Erweiterung *.obj* sind jetzt überflüssig und können gelöscht werden. Dateien mit der Erweiterung *.lib* sind jedoch sehr wichtig und sollten erhalten bleiben. Diese parallelen Bibliotheken sind Dateien, die QuickBASIC zur Erstellung ausführbarer Dateien aus dem Programm verwendet.

Anhand der Programme LINK und LIB können sowohl Quick-Bibliotheken als auch selbständige (*.lib*) Bibliotheken aus der Befehlszeile einer Stapelverarbeitungsdatei erstellt werden. Falls Sie in QuickBASIC ursprünglich in anderen Sprachen geschriebene und kompilierte Routinen einsetzen möchten, müssen Sie zunächst die anderssprachigen Routinen aus der Befehlszeile in einer Quick-Bibliothek ablegen. Sobald sich die anderssprachigen Routinen in der Bibliothek befinden, können auch die BASIC-Module aus der Befehlszeile oder aus der QuickBASIC-Umgebung eingebunden werden.

Ein professioneller Software-Entwickler sollte dem Kunden sowohl die Quick- (*.qlb*) als auch die selbständige (*.lib*) Version der Bibliothek liefern. Ohne die *.lib*-Bibliotheken sind Endnutzer nicht in der Lage, die Bibliotheksroutinen in ausführbaren Dateien zu verwenden, die mit QuickBASIC erstellt werden.

Bei Anlegen einer Quick-Bibliothek unter Verwendung des Linkers muß die *bqb50.lib* Bibliothek immer nach dem dritten Komma auf der LINK-Befehlszeile oder als Antwort auf die Anfrage "Bibliothek" angegeben werden.

H.8 Verwendung anderssprachiger Routinen in einer Quick-Bibliothek

Um Routinen aus anderen Sprachen in einer Quick-Bibliothek abzulegen, müssen Sie mit vorkompilierten oder vorassemblierten Objektdateien beginnen, die die gewünschten anderssprachigen Routinen enthalten. Zu diesem Zweck eignen sich mehrere andere Sprachen, einschließlich Microsoft® C, Microsoft® Macro Assembler, Microsoft® Pascal, Microsoft® FORTRAN sowie jede andere Sprache, die zu der Microsoft-Sprachenfamilie kompatible Objektdateien anlegt.

H.8.1 Aufbau einer Quick-Bibliothek

Die folgenden Schritte sind typisch für das Anlegen einer Quick-Bibliothek, die Routinen aus anderen Sprachen enthält:

1. Gehen Sie von drei, in FORTRAN, C und Macro Assembler erstellten Modulen aus. Der erste Schritt besteht in der Kompilierung und Assemblierung jedes Moduls mit dem entsprechenden Sprachübersetzer. Diese Operationen erzeugen Objektdateien, die hier *for.obj c.obj* und *asm.obj* genannt werden.

2. Anschließend binden Sie die Objektdateien mit /Q, einer besonderen Option von LINK, die den Linker anweist, eine Quick-Bibliothek wie folgt anzulegen:

   ```
   LINK /Q FOR.OBJ C.OBJ ASM.OBJ, GEMISCH.QLB,,BQB50.LIB;
   ```

   Der Linker interpretiert den Eintrag, der den Namen der Objektdateien folgt, (in diesem Fall *gemisch.qlb*) als den Zieldateinamen, unter dem die gebundenen Module abgelegt werden. Daher ist der Name der Quick-Bibliothek in diesem Fall *gemisch.qlb*.

3. Erstellen Sie nun parallel dazu eine *.lib*-Bibliothek mit denselben zum Aufbau der Quick-Bibliothek verwendeten Objektdateien. In diesem Fall ist der erste Name, der dem LIB-Befehl folgt, der Name der *.lib*-Zielbibliothek:

   ```
   LIB GEMISCH.LIB+FOR.OBJ+C.OBJ+ASM.OBJ;
   ```

   Dieser Schritt 3 ist beim Aufbau einer Bibliothek mit anderssprachigen Routinen leicht zu übersehen; dieser Schritt ist jedoch entscheidend, wenn die Bibliothek zur Erstellung einer selbständig ausführbaren Datei eingesetzt werden soll. Ohne diese parallelen selbständigen (*.lib*) Bibliotheken kann QuickBASIC keine ausführbare Datei anlegen, die Routinen der Bibliotheken enthält.

4. Mit den in einer Quick-Bibliothek vorhandenen anderssprachigen Routinen, sowie mit den in einer gleichnamigen selbständigen Bibliothek vorhandenen ursprünglichen Objektdateien können Sie in die QuickBASIC-Umgebung zurückkehren und eine beliebige, nur vom verfügbaren Speicherplatz beschränkte Anzahl von BASIC-Modulen in die Bibliothek einbinden.

Eine eingehende Beschreibung von LINK und LIB finden Sie in Anhang G, "Kompilieren und Binden aus DOS".

H.8.2 Quick-Bibliotheken mit führenden Nullen im ersten Code-Segment

Eine Quick-Bibliothek, die im ersten Code-Segment führende Nullen enthält, ist ungültig und der Versuch diese in QuickBASIC zu laden, hat das Erscheinen der Fehlermeldung `Fehler beim Laden der Datei` *Dateiname* `- ungültiges Format` zur Folge. Das kann beispielsweise vorkommen, wenn die Routine einer Assembler-Sprache auf Null initialisierte Daten im ersten Code-Segment ablegt und diese anschließend beim Anlegen einer Quick-Bibliothek an erster Stelle in der LINK-Befehlszeile aufgeführt ist. Dieses Problem läßt sich wie folgt beheben:

1. Binden Sie zuerst mit Hilfe eines BASIC-Moduls auf der LINK-Befehlszeile.
2. Vergewissern Sie sich, daß das erste Code-Segment des ersten in der LINK-Befehlszeile vorhandenen Moduls kein Null-Byte enthält.

H.8.3 Die Routine B_OnExit

QuickBASIC bietet eine als B_OnExit bezeichnete Funktion auf BASIC-Systemebene. Diese Funktion kommt zur Anwendung, wenn die besonderen Aktionen anderssprachiger Routinen vor (absichtlichem oder unabsichtlichem) Beenden oder Wiederablauf des Programms rückgängig gemacht werden müssen. Zum Beispiel kann es vorkommen, daß ein aktuelles Programm innerhalb der QuickBASIC-Umgebung, das anderssprachige Routinen in einer Quick-Bibliothek aufruft, nicht immer normal beendet wird. Wenn solche Routinen bei Beendigung besondere Aktionen (wie Löschen zuvor installierter Unterbrechungsvektoren) erfordern, gewährleistet der eingeschlossene Abruf von B_OnExit in der Routine, daß die erforderlichen Beendigungs-Routinen stets aufgerufen werden. Nachstehendes Beispiel veranschaulicht einen Aufruf dieser Art (der Fehlerbehandlungs-Code wurde einfachheitshalber ausgelassen). Diese Funktion würde in C in einem Large-Modell kompiliert sein.

```
#include <malloc.h>
extern pascal far B_OnExit ();   /* Deklariere die Routine */
int  *p_IntDatf;
void InitProz ()
{
   void EndeProz ();  /* Deklariere die Funktion EndeProz */
   /* Reserviere weiten (far) Speicherplatz für ein       */
   /* Datenfeld */ mit 20 Ganzzahlen                      */
```

```
   p_IntDatf = (int *)malloc(20*sizeof(int));
   /* Verbinde die Beendigungsroutine (EndeProz) mit      */
   /* BASIC:                                               */
     B_OnExit(EndeProz);
}
                      /* Die Funktion EndeProz wird       */
void EndeProz()       /* Wiederaufruf oder Beendigung     */
{                     /* des Programms aufgerufen         */
   free (p_IntDatf);  /* Gib Speicherplatz der zuvor      */
                      /* von InitProz reserviert wur-     */
}                     /* de frei                          */
```

Würde sich die Funktion `InitProz` in einer Quick-Bibliothek befinden, würde B_OnExit im Fall eines Programmabsturzes die geeignete Freigabe des mit dem Aufruf von `malloc` reservierten Speicherplatzes bewirken. Da das Programm öfters aus der QuickBASIC-Umgebung ausgeführt werden kann, läßt sich diese Routine ebenfalls öfters aufrufen. Die Funktion `EndeProz` selbst kann jedoch nur ein einziges Mal während eines Programmablaufes aufgerufen werden.

Das folgende BASIC-Programm verdeutlicht den Aufruf einer Funktion `InitProz`:

```
DECLARE SUB InitProz CDECL
X = SETMEM(-2048)          ' Mache Platz für die Zuweisung
                           ' des Speicherplatzes mit malloc
                           ' in der C-Funktion.
CALL InitProz
END
```

Wenn über 32 Routinen eingetragen werden, gibt B_OnExit NULL zurück als Hinweis dafür, daß der Speicherplatz für die Eintragung der aktuellen Routine nicht mehr ausreicht. (B_OnExit hat dieselben Rücksprungswerte wie die OnExit-Routine der Microsoft C Laufzeitbibliothek.)

B_OnExit kann mit jeder jeweils in der Quick-Bibliothek plazierten anderssprachigen Routine (einschließlich die der Assembler-Sprache) verwendet werden. Bei vollständig aus der Befehlzeile kompilierten und gebundenen Programmen ist B_OnExit optional.

H.9 Quick-Bibliotheken und Speicherplatz

Da eine Quick-Bibliothek im wesentlichen eine ausführbare Datei darstellt (obwohl sie nicht selbst von der DOS-Befehlszeile aufgerufen werden kann), ist sie im Vergleich zur Summe ihrer Quelldateien ziemlich umfangreich. Dadurch wird die Anzahl der in einer Quick-Bibliothek abgelegten Routinen begrenzt. Um die maximale Größe der Quick-Bibliothek zu bestimmen, addieren Sie den Speicherplatz, den DOS, *qb.exe* und das Hauptmodul Ihres Programmes benötigen. Eine einfache Methode zur Berechnung dieser Faktoren besteht im Einschalten der Maschine, Starten des Programms mit QuickBASIC und der anschließenden Eingabe dieses Befehls in das Direkt-Fenster:

```
PRINT FRE (-1)
```

Dieser Befehl gibt die Anzahl freier Speicherbytes und somit die maximale Größe einer mit diesem Programm verknüpften Quick-Bibliothek an. In den meisten Fällen ist der für eine Quick-Bibliothek erforderliche Speicherplatz ungefähr gleich der Größe ihrer Diskettendatei. Eine Ausnahme zu dieser Faustregel bildet eine Bibliothek mit Prozeduren, die zahlreiche Zeichenketten verwenden und möglicherweise mehr Speicherplatz benötigen.

H.10 Erstellen kompakter ausführbarer Dateien

Wie oben erläutert, erstellt QuickBASIC beim Anlegen einer Quick-Bibliothek auch eine selbständige (*.lib*) Bibliothek mit Objektmodulen, in der jedes Objektmodul einem der Module in der Quick-Bibliothek entspricht. Beim Anlegen einer ausführbaren Datei durchsucht QuickBASIC die selbständige *.lib*-Bibliothek nach Objektmodulen, die die Prozeduren enthalten, auf die in dem Programm Bezug genommen wird.

Enthält ein Objektmodul keine der in dem Programm aufgeführten Prozeduren, wird es nicht in die ausführbare Datei eingebunden. Ein einzelnes Modul kann jedoch zahlreiche Prozeduren enthalten und selbst wenn nur eine davon in dem Programm erscheint, werden sämtliche Prozeduren in die ausführbare Datei aufgenommen. Selbst wenn ein Programm nur eine von vier Prozeduren eines bestimmten Moduls verwendet, wird das gesamte Modul Bestandteil der endgültigen ausführbaren Datei.

Um die ausführbaren Dateien so kompakt wie möglich zu gestalten, sollten Sie eine Bibliothek aufbauen, in der jedes Modul nur direkt zusammenhängende Prozeduren enthält. Im Zweifelsfall können Sie den Inhalt der Bibliothek mit dem in Abschnitt H.4.3, "Anzeigen des Inhalts einer Quick-Bibliothek", beschriebenen Hilfsprogramm *qbibdru.bas* auflisten.

Anhang I: Fehlermeldungen

Während der Entwicklung eines BASIC-Programmes mit QuickBASIC können folgende Fehlerarten auftreten:

- Aufruffehler
- Kompilierzeitfehler
- Linkzeitfehler
- Laufzeitfehler

Jeder Fehlertyp ist mit einem bestimmten Schritt im Entwicklungsprozeß eines Programmes verbunden. Aufruffehler treten beim Aufrufen von QuickBASIC mit den Befehlen QB oder BC auf. Kompilierzeitfehler und -warnungen treten während der Kompilierung, Laufzeitfehler während der Ausführung des Programmes auf. Linkzeitfehler treten nur auf, wenn Sie den Befehl LINK verwenden, um mit BC oder anderen Sprach-Compilern erstellte Objektdateien zu binden.

Abschnitt I.2 listet die Aufruf-, Kompilierzeit- und Laufzeitfehlermeldungen alphabetisch mit den jeweils zugewiesenen Fehlercodes auf. Tabelle I.1 führt die Laufzeitfehlermeldungen und -fehlercodes in numerischer Reihenfolge auf. Abschnitt I.3 listet die Fehlermeldungen des Microsoft Overlay-Linkers und Abschnitt I.4 die Fehlermeldungen des Microsoft-Bibliotheksmanagers auf.

I.1 Anzeige der Fehlermeldungen

Beim Auftreten eines Laufzeitfehlers innerhalb der QuickBASIC-Umgebung (mit Standard-Bildschirmoptionen) erscheint die Fehlermeldung in einem Dialogfeld und der Cursor wird in der Zeile positioniert, in der der Fehler aufgetreten ist.

In selbständig ausführbaren Programmen (die durch Eingabe des Basisnamens der ausführbaren Datei bei der Systemanfrage ausgeführt werden), gibt das Laufzeitsystem die von einer Adresse gefolgten Fehlermeldungen aus, sofern nicht eine der Optionen /D, /E oder /W in der BC-Befehlszeile angegeben wird. In diesen Fällen folgt der Fehlermeldung außerdem die Nummer der Zeile, in der der Fehler aufgetreten ist. Dieser Typ von Fehlermeldungen hat folgende Standardformate:

`Fehler` *n* `in Modul` *Modulname* `bei Adresse` *Segment:Offset*

und

`Fehler` *n* `in Zeile` *Zeilennummer* `von Modul` *Modulname* `bei Adresse` *Segment:Offset*.

Bei einigen Fehlern wird ein "**ERR**-Code" aufgelistet. Tritt ein Fehler auf, wird der **ERR**-Wert beim Aufruf der Fehlerverfolgungs-Unterroutine auf den entsprechenden Code gesetzt. (Fehlerverfolgungs-Routinen werden über die Anweisung **ON ERROR** eingegeben.) Der **ERR**-Wert bleibt unverändert, bis eine **RESUME**-Anweisung die Kontrolle an das Hauptprogramm zurückgibt. Weitere Informationen finden Sie in Kapitel 6, "Fehler- und Ereignisverfolgung".

Tabelle I.1 listet die Fehlercodes in numerischer Reihenfolge auf. Erläuterungen zu den Fehlern finden Sie in der alphabetischen Auflistung.

Tabelle I.1 Laufzeitfehlermeldungen

Code	Beschreibung
2	Syntaxfehler
3	RETURN ohne GOSUB
4	Außerhalb von DATA
5	Unzulässiger Funktionsaufruf
6	Überlauf
7	Speicherkapazität reicht nicht aus
9	Index außerhalb des Bereichs
10	Doppelte Definition
11	Division durch Null
13	Unverträgliche Datentypen
14	Platz für Zeichenkette nicht ausreichend
16	Zeichenkettenformel zu umfangreich
19	RESUME fehlt
20	RESUME ohne Fehler
24	Laufzeitfehler am Gerät
25	Gerätefehler
27	Papier zu Ende
39	CASE ELSE erwartet
40	Variable erforderlich
50	FIELD-Überlauf
51	Interner Fehler
52	Falscher Dateiname oder falsche Dateinummer
53	Datei nicht gefunden
54	Falscher Dateimodus
55	Datei bereits geöffnet
56	FIELD-Anweisung aktiv
57	Geräte-E/A-Fehler
58	Datei existiert bereits
59	Falsche Datensatzlänge
61	Diskette/Festplatte voll
62	Eingabe nach Dateiende
63	Falsche Datensatznummer
64	Unzulässiger Dateiname
67	Zu viele Dateien
68	Gerät nicht verfügbar
69	Überlauf des Kommunikationspuffers
70	Zugriff nicht gestattet
71	Diskette/Festplatte nicht bereit
72	Datenträgerfehler
73	Erweiterte Eigenschaft verfügbar nicht
74	Umbenennen zwischen Disketten/Festplatte
75	Pfad/Datei-Zugriffsfehler falsche Dateinummer
76	Pfad nicht gefunden

1.2 Aufruf-, Kompilierzeit- und Laufzeitfehlermeldungen

`"FN" am Anfang nicht möglich`

Sie haben "FN" als die ersten beiden Buchstaben eines Unterprogramm- oder Variablennamens benutzt. "FN" kann für die ersten beiden Buchstaben nur beim Aufruf einer **DEF FN**-Funktion verwendet werden. (Kompilierzeitfehler)

`$INCLUDE-Datei-Zugriffsfehler`

Die im Metabefehl **$INCLUDE** genannte Include-Datei kann nicht gefunden werden. (Kompilierzeitfehler)

`$Metabefehl-Fehler`

Ein Metabefehl ist falsch. Beim Kompilieren des Programms mit BC ist dieser Fehler nicht "fatal"; das Programm wird zwar ausgeführt, die Ergebnisse können aber falsch sein. (Kompilierzeitwarnung)

`/C: Puffergröße zu groß`

Die maximale Größe des Kommunikationspuffers beträgt 32.767 Bytes. (BC-Aufruffehler)

`ALIAS erfordert Zeichenkettenkonstante`

Das Schlüsselwort **ALIAS** der Anweisung **DECLARE** erfordert als Argument eine Zeichenkettenkonstante. (Kompilierzeitfehler)

`Angabe eines unzulässigen formalen Parameters`

Es liegt ein Fehler in der Parameterliste einer Funktion oder eines Unterprogramms vor. (Kompilierzeitfehler)

`Anweisung ignoriert`

Sie verwenden den BC-Befehl zum Kompilieren eines Programmes, das **TRON**- und **TROFF**-Anweisungen enthält, ohne die Option /D zu benutzen. Dieser Fehler ist nicht "fatal"; das Programm wird zwar ausgeführt, die Ergebnisse können aber falsch sein. (Kompilierzeitwarnung)

`Anweisung in $INCLUDE-Datei unzulässig`

SUB...END SUB- und **FUNCTION...END FUNCTION**-Anweisungsblöcke sind in Include-Dateien nicht zulässig. Benutzen Sie den Befehl **Zusammenführen** aus dem Menü **Datei** zum Einfügen der Include-Datei in das aktuelle Modul, oder laden Sie die Include-Datei als ein separates Modul. Das Laden der Include-Datei als ein separates Modul, kann einige Umstrukturierungen erfordern, da gemeinsam benutzte Variablen nur innerhalb eines Modulbereichs gemeinsam benutzt werden können. (Kompilierzeitfehler)

`Anweisung in TYPE-Block unzulässig`

REM und *Element* **AS** *Typname* sind die einzigen Anweisungen, die zwischen **TYPE**- und **END TYPE** stehen dürfen. (Kompilierzeitfehler)

`Anweisung kann nicht vor SUB/FUNCTION-Definition stehen`

REM und **DEF***Typ* sind die einzigen Anweisungen, die vor einer Prozedurdefinition stehen dürfen. (Kompilierzeitfehler)

`Anweisung muß am Zeilenanfang stehen`

In Block-**IF...THEN...ELSE**-Konstruktionen darf vor **IF**, **ELSE**, **ELSEIF** und **END IF** nur eine Zeilennummer oder eine Zeilenmarke stehen. (Kompilierzeitfehler)

`Anweisungen/Marken zwischen SELECT CASE und CASE unzulässig`

Anweisungen und Zeilenmarken sind zwischen **SELECT CASE** und der ersten **CASE**-Anweisung nicht zulässig. Erlaubt sind Kommentare und Anweisungstrenner. (Kompilierzeitfehler)

`AS-Klausel bei erster Deklaration erforderlich`

Auf eine Variable, die nicht unter Verwendung einer **AS**-Klausel deklariert ist, wird in einer **AS**-Klausel Bezug genommen. (Kompilierzeitfehler)

`AS-Klausel erforderlich`

Es wird auf eine mit einer **AS**-Klausel deklarierte Variable ohne diese Klausel Bezug genommen. Falls die erste Deklaration einer Variablen eine **AS**-Klausel hat, muß jede folgende **DIM, REDIM, SHARED** und **COMMON**-Anweisung, die Bezug auf diese Variable nimmt, eine **AS**-Klausel enthalten. (Kompilierzeitfehler)

`Ausdruck zu komplex`

Dieser Fehler wird durch Überschreitung bestimmter interner Begrenzungen verursacht. Zum Beispiel werden während der Auswertung eines Ausdrucks Zeichenketten, die nicht mit Variablen verknüpft sind, temporäre Adressen zugewiesen. Dieser Fehler kann infolge zahlreicher solcher Zeichenketten auftreten. Versuchen Sie, die Ausdrücke zu vereinfachen und Zeichenketten Variablen zuzuweisen. (Kompilierzeitfehler)

`Außerhalb des Datenbereiches`

Versuchen Sie, den erforderlichen Datenbereich wie folgt zu verändern:

- Verwenden Sie in der **LEN**-Klausel der Anweisung **OPEN** einen kleineren Dateipuffer.
- Verwenden Sie für das Anlegen dynamischer Datenfelder den Metabefehl **$DYNAMIC**. Daten dynamischer Datenfelder können normalerweise viel größer als Daten statischer Datenfelder sein.
- Verwenden Sie anstelle von Datenfeldern mit Zeichenketten variabler Länge Datenfelder mit Zeichenketten fester Länge.
- Verwenden Sie den kleinsten geeigneten Datentyp. Verwenden Sie so weit wie möglich Ganzzahlen.
- Versuchen Sie nicht, viele kleine Prozeduren zu verwenden. QuickBASIC muß zu jeder Prozedur einige Bytes Kontrollinformationen erstellen.
- Verändern Sie die Größe des Stapels (Stack) durch **CLEAR**. Verwenden Sie nicht mehr Stapelplatz als erforderlich.
- Verwenden Sie keine Quellzeilen, die länger als 256 Zeichen sind. Solche Zeilen erfordern die Zuweisung von zusätzlichem Textpufferbereich.

(Kompilierzeit- oder Laufzeitfehler)

`Außerhalb des Stapelbereiches`

Dieser Fehler kann auftreten, wenn eine rekursive **FUNCTION** zu tief verschachtelt ist oder zu viele aktive Aufrufe von Unterroutinen, **SUBs** und **FUNCTIONs** existieren. Der dem Programm zugewiesene Stapelplatz kann mit Hilfe der **CLEAR**-Anweisung vergrößert werden. Dieser Fehler kann nicht verfolgt werden. (Laufzeitfehler)

`Außerhalb von DATA`

Es wird eine **READ**-Anweisung ausgeführt, obwohl in dem Programm keine **DATA**-Anweisungen mit ungelesenen Daten übrig sind. (Laufzeitfehler)

ERR-Code: 4

`Benutzerdefinierter Variablentyp in Ausdruck unzulässig`

Variablen mit benutzerdefiniertem Typ sind in Ausdrücken wie `CALL ALPHA ((X))` nicht zulässig, wobei `X` ein benutzerdefinierter Typ ist. (Kompilierzeitfehler)

`Bezeichner dürfen keinen Punkt enthalten`

Bezeichner von benutzerdefinierten Typen und Namen von Datensatzelementen dürfen keine Punkte enthalten. Der Punkt sollte nur als Datensatz-Variablentrenner benutzt werden. Zusätzlich darf ein Variablenname keinen Punkt enthalten, wenn der Teil des Namens vor dem Punkt irgendwo im Programm in einer Klausel *Bezeichner* **AS** *Benutzertyp* verwendet wird. Wenn Sie ein Programm benutzen, das den Punkt in Variablennamen verwendet, ist es ratsam, diesen durch Groß-/Kleinschreibung zu ersetzen. Aus der Variablen `ALPHA.BETA` beispielsweise würde `AlphaBeta` werden. (Kompilierzeitfehler)

`Bezeichner erwartet`

Sie versuchen, eine Zahl oder ein BASIC-reserviertes Wort zu benutzen, während ein Bezeichner erwartet wird. (Kompilierzeitfehler)

`Bezeichner kann nicht mit %, &, !, # oder $ enden`

Die oben aufgeführten Zusätze sind nicht in Typbezeichnern, Unterprogrammnamen oder benannten **COMMON**-Namen erlaubt. (Kompilierzeitfehler)

`Bezeichner zu lang`

Bezeichner dürfen nicht länger als 40 Zeichen sein. (Kompilierzeitfehler)

`Binäre Quelldatei`

Die Datei, die Sie zu kompilieren versucht haben, ist keine ASCII-Datei. Alle von BASICA gespeicherten Quelldateien sollten mit der Option ,A gespeichert werden. QuickBASIC benutzt diese Meldung ebenfalls, wenn Sie versuchen, die Code-View-Optionen /ZI oder /ZD mit binären Quelldateien zu benutzen. (Kompilierzeitfehler)

`Block-IF ohne END IF`

In einer Block-**IF**-Konstruktion fehlt das zugehörige **END IF**. (Kompilierzeitfehler)

`BYVAL gestattet nur numerische Argumente`

BYVAL akzeptiert keine Zeichenketten- oder Datensatzargumente. (Kompilierzeitfehler)

`CASE ELSE erwartet`

Für einen Ausdruck in einer **SELECT CASE**-Anweisung wurde kein passender Fall gefunden. (Laufzeitfehler)

ERR-Code: 39

`CASE ohne SELECT`

Der erste Teil einer **SELECT CASE**-Anweisung fehlt oder ist falsch geschrieben. (Kompilierzeitfehler)

`COMMON in Quick-Bibliothek zu klein`

In dem Modul sind mehr gemeinsame Variablen angegeben als in der aktuell geladenen Quick-Bibliothek. (Kompilierzeitfehler)

`COMMON und DECLARE müssen vor ausführbaren Anweisungen stehen`

Eine **COMMON**- oder **DECLARE**-Anweisung ist falsch plaziert. **COMMON**- und **DECLARE**-Anweisungen müssen vor der ersten ausführbaren Anweisung stehen. Alle BASIC-Anweisungen, bis auf die folgenden, sind ausführbar:

- **COMMON**
- **DEF***Typ*
- **DIM** (für statische Datenfelder)
- **OPTION BASE**
- **REM**
- **TYPE**
- Alle Metabefehle

(Kompilierzeitfehler)

`CONST/DIM SHARED folgt auf SUB/FUNCTION`

Die Anweisungen **CONST** und **DIM SHARED** sollten vor jeglicher Definition eines Unterprogrammes oder **FUNCTION**-Prozedur erscheinen. Beim Kompilieren des Programms mit BC, ist dieser Fehler nicht "fatal"; das Programm wird zwar ausgeführt, die Ergebnisse können aber falsch sein. (Kompilierzeitwarnung)

`Datei (Dateiname) nicht gefunden. Pfad eingeben:`

Dieser Fehler tritt auf, wenn QuickBASIC eine Quick-Bibliothek oder ein Quick-Dienstprogramm (*bc.exe*, *link.exe*, *lib.exe*, oder *qb.exe*), die/das vom Programm benötigt wird, nicht finden kann. Geben Sie den Pfad-Namen korrekt ein oder betätigen Sie STRG+C, um zur DOS-Anfrage zurückzukehren. (QB-Aufruffehler)

`Datei bereits geöffnet`

Für eine bereits geöffnete Datei wird eine **OPEN**-Anweisung im sequentiellen Ausgabemodus, oder eine **KILL**-Anweisung angegeben. (Laufzeitfehler)

ERR-Code: 55

`Datei existiert bereits`

Der in einer **NAME**-Anweisung angegebene Dateiname ist mit einem bereits auf der Diskette/Festplatte verwendeten Dateinamen identisch. (Laufzeitfehler)

ERR-Code: 58

`Datei ist bereits geladen`

Sie versuchen, eine Datei zu laden, die sich bereits im Speicher befindet. (Kompilierzeitfehler)

`Datei nicht gefunden`

Eine Anweisung **KILL, NAME, FILES** oder **OPEN** bezieht sich auf eine Datei, die an der spezifizierten Stelle nicht existiert. (Laufzeitfehler)

ERR-Code: 53

`Datei nicht gefunden; im Modul` *Modulname* `bei Adresse` *Segment:Offset*

Eine Anweisung **FILES**, **KILL**, **NAME**, **OPEN** oder **RUN** bezieht sich auf eine nicht vorhandene Datei. Diese Fehlermeldung entspricht der Meldung `Datei nicht gefunden,` mit dem Unterschied, daß sie während der Ausführung eines kompilierten Programmes erscheint. Die Adresse ist die Fehlerdateilage im Code. (Laufzeitfehler)

ERR-Code: 53

`Datenfeld bereits dimensioniert`

Dieser Fehler kann aus folgenden Gründen auftreten:

- Mehr als eine **DIM**-Anweisung für dasselbe statische Datenfeld.
- Eine **DIM**-Anweisung nach erster Verwendung eines Datenfeldes. Die Zuordnung für statische Datenfelder muß mit der Anweisung **ERASE** aufgehoben werden, bevor die Felder neu dimensioniert werden können; dynamische Datenfelder können ebenfalls mit der Anweisung **REDIM** erneut dimensioniert werden.
- Eine **OPTION BASE**-Anweisung erscheint nach Dimensionierung eines Datenfeldes.

(Kompilierzeit- oder Laufzeitfehler)

`Datenfeld ist nicht definiert`

Es wird auf ein Datenfeld Bezug genommen, das nicht definiert ist. (Kompilierzeitfehler)

`Datenfeld ist nicht dimensioniert`

Es wird auf ein nicht dimensioniertes Datenfeld Bezug genommen. Beim Kompilieren des Programms mit BC, ist dieser Fehler nicht "fatal"; das Programm wird zwar ausgeführt, die Programmergebnisse können aber falsch sein. (Kompilierzeitwarnung)

`Datenfeld zu groß`

Der Platz für Benutzerdaten reicht für die Deklaration des Datenfeldes nicht aus. Verkleinern Sie das Datenfeld oder benutzen Sie den **$DYNAMIC**-Metabefehl. Dieser Fehler tritt ebenfalls auf, wenn die Größe des Datenfeldes 64K überschreitet, das Datenfeld nicht dynamisch ist und die /AH-Option nicht verwendet wurde. Reduzieren Sie die Größe des Datenfeldes oder machen Sie das Datenfeld dynamisch und benutzen Sie die /AH-Befehlszeilenoption. (Kompilierzeitfehler)

`Datenträgerfehler`

Die Hardware des Disketten-/Festplattenlaufwerks hat einen physikalischen Defekt auf der Diskette/Festplatte entdeckt. (Laufzeitfehler)

ERR-Code: 72

`Datensatz/Zeichenketten-Zuweisung erforderlich`

Der Anweisung **LSET** fehlt die Zuweisung der Zeichenketten- oder Datensatzvariablen. (Kompilierzeitfehler)

`DECLARE erforderlich`

Ein impliziter Aufruf einer **SUB** oder **FUNCTION** erscheint vor der Definition der Prozedur. (Ein impliziter Aufruf verwendet nicht die Anweisung **CALL**.) Alle Prozeduren müssen vor dem impliziten Aufruf definiert oder deklariert sein. (Kompilierzeitfehler)

`DEF FN in Steuerungsanweisungen nicht erlaubt`

Definitionen von **DEF FN**-Funktionen sind innerhalb von Steuerungsstrukturen wie **IF...THEN...ELSE** und **SELECT CASE** nicht erlaubt. (Kompilierzeitfehler)

`DEF ohne END DEF`

In einer mehrzeiligen Funktionsdefinition gibt es kein zugehöriges **END DEF**. (Kompilierzeitfehler)

`DEFTyp-Zeichenangabe unzulässig`

Eine **DEF***Typ*-Anweisung ist falsch eingegeben. Auf **DEF** kann nur **DBL**, **INT**, **LNG**, **SNG**, **STR** oder (bei benutzerdefinierten Funktionen) ein Leerzeichen folgen. (Kompilierzeitfehler)

`Diskette nicht bereit`

Das Laufwerk ist nicht verriegelt, oder es befindet sich keine Diskette im Laufwerk.

ERR-Code: 71

`Diskette/Festplatte voll`

Auf der Diskette/Festplatte ist nicht genügend Speicherplatz zum Abschluß einer **PRINT**-, **WRITE**- oder **CLOSE**-Operation. Dieser Fehler kann ebenfalls auftreten, wenn QuickBASIC nicht über genügend Platz für die Erstellung einer ausführbaren oder Objekt-Datei verfügt. (Laufzeitfehler)

ERR-Code: 61

`Division durch Null`

In einem Ausdruck wird eine Division durch Null vorgenommen oder Null wird mit einem negativen Wert potenziert. (Kompilierzeit- oder Laufzeitfehler)

ERR-Code: 11

`DO ohne LOOP`

In einer **DO...LOOP**-Anweisung fehlt die abschließende **LOOP**-Klausel. (Kompilierzeitfehler)

`Dokument zu groß`

Das Dokument überschreitet die interne Grenze von QuickBASIC. Teilen Sie das Dokument in separate Dateien auf.

`Doppelpunkt nach /C erwartet`

Zwischen der Option und dem Puffergrößenargument ist ein Doppelpunkt erforderlich. (BC-Aufruffehler)

`Doppelte Definition`

Sie benutzen einen bereits definierten Bezeichner. Sie versuchen beispielsweise, denselben Namen sowohl für eine **CONST**-Anweisung als auch für eine Variable, oder denselben Namen für eine Prozedur und eine Variable zu benutzen.

Dieser Fehler tritt ebenfalls beim Neudimensionieren eines Datenfeldes auf. Eine Neudimensionierung dynamischer Datenfelder ist anhand von **DIM** oder **REDIM** vorzunehmen. (Kompilierzeit- oder Laufzeitfehler)

ERR-Code: 10

`Doppelte Marke`

Zwei Programmzeilen wird dieselbe Nummer oder Marke zugewiesen. Jede Zeilennummer oder Zeilenmarke darf in einem Modul nur einmal vorkommen. (Kompilierzeitfehler)

`DOS 2.10 oder höhere Version erforderlich`

Sie versuchen, QuickBASIC mit einer falschen DOS-Version zu verwenden. (QB-Aufruf- oder Laufzeitfehler)

`Dynamisches Datenfeldelement unzulässig`

Dynamische Datenfeldelemente sind mit **VARPTR$** nicht erlaubt. (Kompilierzeitfehler)

`Einfache Variable oder Datenfeldvariable erwartet`

QuickBASIC erwartet ein Variablenargument. (Kompilierzeitfehler)

`Eingabe nach Dateiende`

Eine **INPUT**-Anweisung liest aus einer leeren oder aus einer Datei ein, aus der sämtliche Daten bereits gelesen wurden. Verwenden Sie zur Vermeidung dieses Fehlers die Funktion **EOF**, um das Dateiendezeichen zu ermitteln. (Laufzeitfehler)

ERR-Code: 62

`Eingegebene Datei nicht gefunden`

Die auf der Befehlszeile angegebene Quelldatei befindet sich nicht an der angegebenen Stelle. (BC-Aufruffehler)

`Element nicht definiert`

Es wird auf ein nicht definiertes Element eines benutzerdefinierten Typs Bezug genommen. Wenn beispielsweise der benutzerdefinierte Typ `MEINTYP` die Elemente `A`, `B` und `C` enthält, würde der Versuch, die Variable `D` als ein Element von `MEINTYP` zu verwenden, zu dieser Fehlermeldung führen. (Kompilierzeitfehler)

`ELSE ohne IF`

Eine **ELSE**-Klausel erscheint ohne ein dazugehöriges **IF**. Manchmal entsteht dieser Fehler durch falsches Verschachteln von **IF**-Anweisungen. (Kompilierzeitfehler)

`ELSEIF ohne IF`

Eine **ELSEIF**-Anweisung erscheint ohne ein dazugehöriges **IF**. Manchmal entsteht dieser Fehler durch falsches Verschachteln von **IF**-Anweisungen. (Kompilierzeitfehler)

`END DEF ohne DEF`

Eine Anweisung **END DEF** hat keine zugehörige Anweisung **DEF**. (Kompilierzeitfehler)

`END IF ohne Block-IF`

Der Beginn eines **IF**-Blockes fehlt. (Kompilierzeitfehler)

`END SELECT ohne SELECT`

Das Ende einer **SELECT CASE**-Anweisung erscheint ohne ein einleitendes **SELECT CASE**. Der Beginn der **SELECT CASE**-Anweisung könnte fehlen oder falsch geschrieben sein. (Kompilierzeitfehler)

`END SUB oder END FUNCTION muß letzte Zeile im Fenster sein`

Sie versuchen, Code nach einer Prozedur einzufügen. Sie müssen entweder zum Hauptmodul zurückkehren oder ein anderes Modul öffnen. (Kompilierzeitfehler)

`END SUB/FUNCTION ohne SUB/FUNCTION`

Sie haben die **SUB**- oder **FUNCTION**-Anweisung gelöscht. (Kompilierzeitfehler)

`END TYPE ohne TYPE`

Eine **END TYPE**-Anweisung wird außerhalb einer **TYPE**-Deklaration benutzt. (Kompilierzeitfehler)

`Erwartet:` *Objekt*

Es handelt sich um einen Syntaxfehler. Der Cursor ist auf dem unerwarteten Objekt positioniert. (Kompilierzeitfehler)

`Erweiterte Eigenschaft nicht verfügbar`

Sie versuchen eine Eigenschaft von QuickBASIC zu verwenden, die mit einer anderen BASIC-Version verfügbar oder von einer späteren DOS-Version unterstützt ist. (Kompilierzeit- oder Laufzeitfehler)

ERR-Code: 73

`EXIT befindet sich nicht innerhalb von FOR...NEXT`

Eine **EXIT FOR**-Anweisung wird außerhalb einer **FOR...NEXT**-Anweisung benutzt. (Kompilierzeitfehler)

`EXIT DO befindet sich nicht innerhalb von DO...LOOP`

Eine **EXIT DO**-Anweisung wird außerhalb einer **DO...LOOP**-Anweisung benutzt. (Kompilierzeitfehler)

`Falsche Anzahl von Dimensionen`

Der Bezug auf ein Datenfeld enthält die falsche Anzahl von Dimensionen. (Kompilierzeitfehler)

`Falsche Datensatznummer`

In einer **PUT**- oder **GET**-Anweisung ist die Nummer des Datensatzes kleiner gleich Null. (Laufzeitfehler)

ERR-Code: 63

`Falsche Datensatzlänge`

Es wird eine **GET**- oder **PUT**-Anweisung ausgeführt, die eine Datensatzvariable angibt, deren Länge nicht zur angegebenen Datensatzlänge in der zugehörigen **OPEN**-Anweisung paßt. (Laufzeitfehler)

ERR-Code: 59

`Falscher Dateimodus`

Dieser Fehler tritt in folgenden Situationen auf:

- Das Programm versucht, **PUT** oder **GET** für eine sequentielle Datei zu verwenden, oder ein **OPEN** mit einem anderen Modus als I, O oder R auszuführen.
- Das Programm versucht, eine **FIELD**-Anweisung für eine Datei zu verwenden, die nicht für Direktzugriff geöffnet ist.
- Das Programm versucht, in eine für Eingabe geöffnete Datei zu schreiben.
- Das Programm versucht, aus einer für Ausgabe oder Erweiterung geöffneten Datei zu lesen.
- QuickBASIC versucht, eine im komprimierten Format gespeicherte Include-Datei zu verwenden. Include-Dateien müssen im Textformat gespeichert werden. Laden Sie die Include-Datei erneut, speichern Sie diese im Textformat, und versuchen Sie, das Programm erneut zu starten.
- Sie versuchen ein unbrauchbares binäres Programm zu laden.

(Laufzeitfehler)

ERR-Code: 54

`Falscher Dateiname oder falsche Dateinummer`

Eine Anweisung oder ein Befehl bezieht sich auf eine Datei mit einem Dateinamen oder einer Dateinummer, die nicht in der **OPEN**-Anweisung angegeben wurde oder die sich außerhalb des bei der Initialisierung angegebenen Bereiches von Dateinummern befindet. (Laufzeitfehler)

ERR-Code: 52

`Far-Heap Konflikt`

Ein Konflikt im Far-Heap kann in folgenden Fällen auftreten:

- Der QB-Compiler unterstützt keine speicherresidenten Programme in DOS.
- Die **POKE**-Anweisung ändert von QuickBASIC verwendete Speicherbereiche. (Dieser Vorgang kann auch den Beschreiber eines dynamischen Zahlendatenfeldes oder einer Zeichenkette fester Länge verändern.)
- Das Programm hat eine anderssprachige Routine aufgerufen, die den von QuickBASIC verwendeten Speicherbereich geändert hat. (Dieser Vorgang kann auch den Beschreiber eines dynamischen Zahlendatenfeldes oder einer Zeichenkette fester Länge verändern.)

(Kompilierzeitfehler)

`Fehlende linke Klammer`

QuickBASIC erwartet eine linke Klammer oder eine **REDIM**-Anweisung versucht, einem Skalar Speicherplatz erneut zuzuweisen. (Kompilierzeitfehler)

`Fehlende Option On Error (/E)`

Beim Kompilieren mit dem BC-Befehl müssen Programme, die **ON ERROR GOTO**-Anweisungen enthalten, mit der Option On Error (/E) kompiliert werden. (Kompilierzeitfehler)

`Fehlende Option Resume Next (/X)`

Beim Kompilieren mit dem BC-Befehl müssen Programme, die die Anweisungen **RESUME**, **RESUME NEXT** oder **RESUME 0** enthalten, mit der Option Resume Next (/X) kompiliert werden. (Kompilierzeitfehler)

`Fehlende rechte Klammer`

QuickBASIC erwartet eine rechte (abschließende) Klammer. (Kompilierzeitfehler)

`Fehlende SUB oder FUNCTION`

Eine **DECLARE**-Anweisung hat keine entsprechende Prozedur. (Kompilierzeitfehler)

`Fehlende Zeilennummer oder Zeilenmarke`

Eine Zeilennummer oder Zeilenmarke fehlt bei einer Anweisung, die dies erfordert, z. B. **GOTO**. (Kompilierzeitfehler)

`Fehlender Stern`

In einer Zeichenkettendefinition eines benutzerdefinierten Typs fehlt der Stern. (Kompilierzeitfehler)

`Fehlendes AS`

Der Compiler erwartet ein **AS**-Schlüsselwort wie in `OPEN "dateiname" FOR INPUT AS #1`. (Kompilierzeitfehler)

`Fehlendes BASE`

QuickBASIC hat hier das Schlüsselwort **BASE**, wie z.B. in **OPTION BASE**, erwartet. (Kompilierzeitfehler)

`Fehlendes Gleichheitszeichen`

QuickBASIC erwartet ein Gleichheitszeichen. (Kompilierzeitfehler)

`Fehlendes GOSUB`

Einer **ON** *Ereignis*-Anweisung fehlt das **GOSUB**. (Kompilierzeitfehler)

`Fehlendes GOTO`

Einer **ON ERROR**-Anweisung fehlt das **GOTO**. (Kompilierzeitfehler)

`Fehlendes INPUT`

Der Compiler erwartet das Schlüsselwort **INPUT**. (Kompilierzeitfehler)

`Fehlendes Komma`

QuickBASIC erwartet ein Komma. (Kompilierzeitfehler)

`Fehlendes Minuszeichen`

QuickBASIC erwartet ein Minuszeichen. (Kompilierzeitfehler)

`Fehlendes Semikolon`

QuickBASIC erwartet ein Semikolon. (Kompilierzeitfehler)

`Fehlendes THEN`

QuickBASIC erwartet das Schlüsselwort **THEN**. (Kompilierzeitfehler)

`Fehlendes TO`

QuickBASIC erwartet das Schlüsselwort **TO**. (Kompilierzeitfehler)

`Fehlendes TYPE`

Einer **END TYPE**-Anweisung fehlt das Schlüsselwort **TYPE**. (Kompilierzeitfehler)

`Fehler beim Laden der Datei (Dateiname) - Datei nicht gefunden`

Dieser Fehler tritt bei der Umleitung von Eingaben in QuickBASIC aus einer Datei auf. Die Eingabedatei befindet sich nicht an der Stelle, die in der Befehlszeile angegeben wurde. (QB-Aufruffehler)

`Fehler beim Laden der Datei (Dateiname) - Disketten-E/A-Fehler`

Dieser Fehler wird von physikalischen Problemen bei Diskettenzugriffen verursacht, z. B. wenn das Laufwerk nicht verriegelt ist. (QB-Aufruffehler)

`Fehler beim Laden der Datei (Dateiname) - DOS-Speicherbereichsfehler`

In den von DOS verwendeten Speicherbereich wurde entweder von einer Assembler-Routine oder mit einer **POKE**-Anweisung geschrieben. (QB-Aufruffehler)

`Fehler beim Laden der Datei (Dateiname) - Speicherplatz zu gering`

Es wird mehr Speicherplatz erfordert als verfügbar ist. Es kann zum Beispiel sein, daß nicht genügend Speicher für die Zuordnung eines Dateipuffers vorhanden ist. Versuchen Sie, die Größe der DOS-Puffer zu reduzieren, alle speicherresidenten Programme oder einige Gerätetreiber zu entfernen. Wenn Sie umfangreiche Datenfelder haben, versuchen Sie, zu Beginn des Programmes einen **$DYNAMIC**-Metabefehl einzufügen. Das Entfernen geladener Dokumente wird ebenfalls Speicherplatz freigeben. (Laufzeitfehler)

`Fehler beim Laden der Datei (Datei) - ungültiges Format`

Sie versuchen, eine Quick-Bibliothek zu laden, die nicht das korrekte Format hat. Dieser Fehler kann beim Versuch auftreten, eine mit einer früheren Version von QuickBASIC erstellte Quick-Bibliothek zu verwenden, beim Versuch, eine Datei zu verwenden, die nicht mit der QuickBASIC-Option **Bibliothek erstellen** oder der Linker-Option /QU verarbeitet wurde, oder beim Versuch, mit QuickBASIC eine selbständige *.lib*-Bibliothek zu laden. (QB-Aufruffehler)

`Fehler kann nicht ausgegeben werden`

Für die vorliegende Fehlersituation gibt es keine Fehlermeldung. Eine **ERROR**-Anweisung ohne definierten Fehlercode kann die Ursache dieser Meldung sein. (Laufzeitfehler)

`Fehler während der QuickBASIC-Initialisierung`

Dieser Fehler kann verschiedene Ursachen haben. Am häufigsten tritt er auf, wenn die Maschine nicht genügend Speicherplatz zum Laden von QuickBASIC hat. Falls Sie eine Benutzerbibliothek laden, versuchen Sie, diese zu verkleinern.

Dieser Fehler tritt auch beim Versuch auf, QuickBASIC mit nicht unterstützter Hardware zu verwenden. (QB-Aufruffehler)

`FIELD-Anweisung aktiv`

Eine **GET**- oder **PUT**-Anweisung bezieht sich auf eine in einer Datei benutzten Datensatzvariable, der bereits Speicherplatz über eine **FIELD**-Anweisung zugewiesen wurde. **GET** oder **PUT** mit einem Datensatzvariablen-Argument dürfen nur für Dateien verwendet werden, für die keine **FIELD**-Anweisungen ausgeführt wurden. (Laufzeitfehler)

ERR-Code: 56

`FIELD-Überlauf`

Eine **FIELD**-Anweisung versucht, mehr Bytes zuzuweisen, als für die Datensatzlänge einer Direktzugriffsdatei angegeben wurden. (Laufzeitfehler)

ERR-Code: 50

`FOR ohne NEXT`

Jede **FOR**-Anweisung muß mit einer entsprechenden **NEXT**-Anweisung übereinstimmen. (Kompilierzeitfehler)

`FOR-Schleifen-Indexvariable bereits belegt`

Dieser Fehler tritt auf, wenn eine Indexvariable in verschachtelten **FOR**-Schleifen mehr als einmal benutzt wird. (Kompilierzeitfehler)

`Formale Parameter nicht eindeutig`

Eine **FUNCTION**- oder **SUB**-Deklaration enthält doppelte Parameter, wie in `SUB HolName (A,B,C,A) STATIC`. (Kompilierzeitfehler)

`Fortfahren nicht möglich`

Während der Fehlerbeseitigung haben Sie eine Änderung vorgenommen, die die weitere Ausführung verhindert. (Kompilierzeitfehler)

`Funktion bereits definiert`

Dieser Fehler tritt auf, wenn eine bereits definierte **FUNCTION** erneut definiert wird. (Kompilierzeitfehler)

`Funktion nicht definiert`

Sie müssen eine **FUNCTION** vor ihrer Verwendung deklarieren oder definieren. (Kompilierzeitfehler)

`Ganzzahl zwischen 1 und 32767 verlangt`

Die Anweisung erfordert ein ganzzahliges Argument. (Kompilierzeitfehler)

`Gerät nicht verfügbar`

Das Gerät, auf das Sie zuzugreifen versuchen, ist nicht angeschlossen oder nicht vorhanden. (Laufzeitfehler)

ERR-Code: 68

`Geräte-E/A-Fehler`

Bei einer Geräte-E/A-Operation ist ein E/A-Fehler aufgetreten. Das Betriebssystem kann die Ausführung nach diesem Fehler nicht fortsetzen. (Laufzeitfehler)

ERR-Code: 57

`Gerätefehler`

Ein Gerät hat einen Hardwarefehler gemeldet. Wenn diese Meldung erfolgt, während Daten in eine Kommunikationsdatei übertragen werden, zeigt sie an, daß die mit der Anweisung **OPEN COM** geprüften Signale innerhalb des angegebenen Zeitraums nicht gefunden wurden. (Laufzeitfehler)

ERR-Code: 25

`GOTO oder GOSUB erwartet`

QuickBASIC erwartet eine **GOTO**- oder **GOSUB**-Anweisung. (Kompilierzeitfehler)

`Gültige Optionen: [RUN] Dat /AH /B /C:Puf /G /NOHI /H /L [lib] /MBF /CMD ZeiKe`

Diese Meldung erscheint beim Aufruf von QuickBASIC anhand einer ungültigen Option. (QB-Aufruffehler)

`Hilfe nicht gefunden`

Die angeforderte Hilfe wurde nicht gefunden und das Programm enthält Fehler welche QuickBASIC daran hindern eine Variablentabelle aufzubauen. Betätigen Sie F5, um die fehlerverursachende Zeile sichtbar zu machen.

`In Prozedur oder DEF FN unzulässig`

Diese Anweisung ist innerhalb einer Prozedur nicht gestattet. (Kompilierzeitfehler)

`Include-Datei zu groß`

Die Include-Datei überschreitet die interne Grenze von QuickBASIC. Teilen Sie die Datei in separate Dateien auf. (Kompilierzeitfehler)

`Index außerhalb des Bereichs`

Auf einen Datensatz wurde mit einem Index, der außerhalb der Datenfelddimensionen liegt, verwiesen, oder es wurde auf ein Element eines nicht dimensionierten dynamischen Datenfeldes zugegriffen. Diese Meldung erfolgt, wenn während des Kompilierens die Option Debug (/D) angegeben wurde. Dieser Fehler kann auch erscheinen, wenn die Größe des Datenfeldes 64K überschreitet, das Datenfeld nicht dynamisch ist und die /AH-Option nicht verwendet wurde. Reduzieren Sie die Größe des Datenfeldes oder machen Sie das Datenfeld dynamisch und benutzen Sie die /AH-Befehlszeilenoption. (Laufzeitfehler)

ERR-Code: 9

`Interner Fehler`

In QuickBASIC ist eine interne Störung aufgetreten. Melden Sie Microsoft Corporation bitte die Bedingungen, unter denen diese Meldung aufgetreten ist, im beiliegenden Formblatt. (Laufzeitfehler).

ERR-Code: 51

`Interner Fehler bei` *xxxx*

In QuickBASIC ist bei Dateilage *xxxx* eine interne Störung aufgetreten. Melden Sie Microsoft die Bedingungen, unter denen diese Meldung aufgetreten ist, bitte im beiliegenden Formblatt.

`Kein Hauptmodul. Wählen Sie Hauptmodul bestimmen aus dem Menü Ausführen.`

Sie versuchen ein Programm nach Entfernen des Hauptmoduls zu starten. Jedes Programm muß ein Hauptmodul enthalten. (Kompilierzeitfehler)

`Keine Zeilennummer in` *Modulname* `bei Adresse` *Segment:Offset*

Dieser Fehler tritt auf, wenn während einer Fehlerverfolgung die Fehleradresse nicht in der Tabelle der Zeilennummern gefunden wird. Dies kommt vor, wenn keine ganzzahligen Zeilennummern zwischen 0 und 65.527 vorhanden sind. Dieser Fehler kann auch auftreten, wenn das Benutzerprogramm versehentlich die Tabelle der Zeilennummern überschrieben hat. Dieser Fehler ist schwerwiegend und kann nicht verfolgt werden. (Laufzeitfehler)

`Korrigieren Sie die Eingabe`

Auf eine **INPUT**-Anfrage haben Sie mit der falschen Anzahl oder den falschen Typen von Objekten geantwortet. Geben Sie die Antwort in der richtigen Form erneut ein. (Laufzeitfehler)

`Laufzeitfehler am Gerät`

Das Programm hat innerhalb einer vorher festgelegten Zeitspanne keine Informationen von einem E/A-Gerät erhalten. (Laufzeitfehler)

ERR-Code: 24

`Lesefehler auf Standardeingabe`

Es tritt während des Lesens von der Konsole oder aus einer umgeleiteten Eingabedatei ein Systemfehler auf. (BC-Aufruffehler)

`Listing für binäre BASIC-Quelldatei kann nicht erstellt werden`

Sie versuchen, eine binäre Quelldatei mit dem Befehl BC und der Option /A zu kompilieren. Kompilieren Sie erneut ohne die Option /A. (BC-Aufruffehler)

`LOOP ohne DO`

Das eine **DO...LOOP**-Anweisung einleitende DO fehlt oder ist falsch geschrieben. (Kompilierzeitfehler)

`Marke nicht definiert`

Es wird auf eine Zeilenmarke Bezug genommen (z. B. in einer **GOTO**-Anweisung), die in dem Programm nicht erscheint. (Kompilierzeitfehler)

`Marke nicht definiert:` *Marke*

Eine Anweisung **GOTO**-*Zeilenmarke* nimmt Bezug auf eine nicht existierende Zeilenmarke. (Kompilierzeitfehler)

`Mathematischer Überlauf`

Das Ergebnis einer Berechnung ist zu groß, um im BASIC-Zahlenformat dargestellt zu werden. (Kompilierzeitfehler)

`Modul nicht gefunden. Modul aus Programm entfernen?`

Beim Laden des Programmes hat QuickBASIC die Datei, die das angegebene Modul enthält, nicht gefunden. QuickBASIC hat statt dessen ein leeres Modul angelegt. Sie müssen nun vor Start dieses Programms das leere Modul löschen.

`Modul-Ebenen-Code zu groß`

Der Modul-Ebenen-Code überschreitet die internen Grenzen von QuickBASIC. Versuchen Sie, einen Teil des Codes in **SUB**- oder **FUNCTION**-Prozeduren unterzubringen. (Kompilierzeitfehler)

`Name des Unterprogramms unzulässig`

Dieser Fehler tritt auf, wenn ein Unterprogrammname ein in BASIC reserviertes Wort ist oder dieser zweimal vergeben wurde. (Kompilierzeitfehler)

`NEXT fehlt für Variable`

Einer **FOR**-Anweisung fehlt die zugehörige **NEXT**-Anweisung. Die Variable ist die **FOR**-Schleifen-Indexvariable. (Kompilierzeitfehler)

`NEXT ohne FOR`

Jede **NEXT**-Anweisung muß eine passende **FOR**-Anweisung haben. (Kompilierzeitfehler)

`Nicht anzeigbar`

Dieser Fehler tritt auf, wenn Sie in einem Anzeigeausdruck eine Variable angeben. Stellen Sie sicher, daß das Modul oder die Prozedur in dem aktiven Arbeitsfenster Zugriff auf die zu beobachtende Variable hat. Zum Beispiel kann im Modul-Ebenen-Code nicht auf Variablen zugegriffen werden, die lokal zu **SUB** oder **FUNCTION** sind. (Laufzeitfehler)

`Nicht erkannter Schalterfehler: "QU"`

Sie versuchen, eine *.exe*-Datei oder Quick-Bibliothek mit einer falschen Version des Microsoft Overlay-Linkers zu erstellen. Sie müssen den mit den Originaldisketten gelieferten Linker verwenden, um eine *.exe*-Datei oder eine Quick-Bibliothek zu erstellen. (Kompilierzeitfehler)

`Numerisches Datenfeld unzulässig`

Numerische Datenfelder sind als Argumente für VARPTR$ nicht zulässig. Nur einfache Variablen und Elemente von Zeichenkettendatenfeldern sind erlaubt. (Kompilierzeitfehler)

`Nur einfache Variablen erlaubt`

In **READ**- und **INPUT**-Anweisungen sind benutzerdefinierte Typen und Datenfelder nicht erlaubt. Datenfeldelemente eines nicht benutzerdefinierten Typs sind zulässig. (Kompilierzeitfehler)

`Operation erfordert Diskette/Festplatte`

Sie versuchen auf bzw. von einem Nicht-Disketten-Gerät zu speichern bzw. zu laden, wie z. B. einem Drucker oder der Tastatur. (Kompilierzeitfehler)

`Option Ereignisver. (/W) oder Prüfen zw. Anweisungen (/V) fehlt`

Das Programm enthält eine **ON** *Ereignis*-Anweisung, die eine dieser Optionen benötigt. (Kompilierzeitfehler)

`Papier zu Ende`

Dem Drucker ist das Papier ausgegangen, oder er ist nicht eingeschaltet. (Laufzeitfehler)

ERR-Code: 27

`Pfad des Laufzeitmoduls eingeben:`

Diese Anfrage erscheint, wenn das Laufzeitmodul *brun45.exe* nicht gefunden wird. Geben Sie den korrekten Pfad ein. Dies ist ein schwerwiegender Fehler, der nicht verfolgt werden kann. (Laufzeitfehler)

`Pfad nicht gefunden`

Während einer Operation mit **OPEN, MKDIR, CHDIR** oder **RMDIR** konnte DOS das angegebene Verzeichnis nicht finden. Die Operation wird nicht beendet. (Laufzeitfehler)

ERR-Code: 76

`Pfad/Datei-Zugriffsfehler falsche Dateinummer`

Während einer Operation mit **OPEN**, **MKDIR**, **CHDIR** oder **RMDIR** konnte DOS keine korrekte Pfad/Dateinamen-Verbindung herstellen. Die Operation wird nicht beendet. (Kompilierzeit- oder Laufzeitfehler)

ERR-Code: 75

`Platz für Zeichenkette nicht ausreichend`

Zeichenkettenvariablen überschreiten den zugeordneten Zeichenkettenbereich. (Laufzeitfehler)

ERR-Code: 14

`Prozedur bereits in Quick-Bibliothek definiert`

Eine Prozedur in der Quick-Bibliothek hat denselben Namen wie eine Prozedur des Programms. (Kompilierzeitfehler)

`Prozedur zu groß`

Diese Prozedur überschreitet die internen Grenzen von QuickBASIC. Verkleinern Sie die Prozedur durch Aufteilung in mehrere Prozeduren. (Kompilierzeitfehler)

`Puffergröße erwartet nach /C:`

Sie müssen eine Puffergröße nach der Option /C angeben. (BC-Aufruffehler)

`RESUME fehlt`

Während das Programm sich in einer Fehlerverfolgungsroutine befand, wurde das Programmende erreicht. Zur Behebung dieser Situation wird eine **RESUME**-Anweisung benötigt. (Laufzeitfehler)

ERR-Code: 19

`RESUME ohne Fehler`

Eine **RESUME**-Anweisung wird vor Aufruf einer Fehlerverfolgungsroutine erreicht. (Laufzeitfehler)

ERR-Code: 20

`RETURN ohne GOSUB`

Es wird eine **RETURN**-Anweisung erreicht, für die es keine vorhergehende passende **GOSUB**-Anweisung gibt. (Laufzeitfehler)

ERR-Code: 3

`SEG oder BYVAL in CALLS nicht erlaubt`

BYVAL und **SEG** sind nur in einer **CALL**-Anweisung zulässig. (Kompilierzeitfehler)

`SELECT ohne END SELECT`

Das Ende einer **SELECT CASE**-Anweisung fehlt oder ist falsch geschrieben. (Kompilierzeitfehler)

`Speicherkapazität reicht nicht aus`

Es wird mehr Speicher benötigt als verfügbar ist. Es kann zum Beispiel nicht genügend Speicher für die Zuweisung eines Dateipuffers vorhanden sein. Versuchen Sie, die Größe der DOS-Puffer zu verkleinern, entfernen Sie speicherresidente Programme, oder einige Gerätetreiber. Wenn Sie umfangreiche Datenfelder haben, versuchen Sie, am Anfang des Programms einen **$DYNAMIC**-Metabefehl einzusetzen. Das Entfernen geladener Dokumente gibt ebenfalls Speicherplatz frei. (BC-Aufruf-, Kompilierzeit- oder Laufzeitfehler)

ERR-Code: 7

`Steuerungsstruktur in IF...THEN...ELSE unvollständig`

Eine nicht passende **NEXT**-, **WEND**-, **END IF**-, **END SELECT**- oder **LOOP**-Anweisung erscheint in einer einzeiligen **IF...THEN...ELSE**-Anweisung. (Kompilierzeitfehler)

`STOP in Modul` *Modulname* `bei Adresse` *Segment:Offset*

Das Programm hat eine **STOP**-Anweisung erreicht. (Laufzeitfehler)

`SUB/FUNCTION ohne END SUB/FUNCTION`

Einer Prozedur fehlt die abschließende Anweisung. (Kompilierzeitfehler)

`Syntaxfehler`

Diesen Fehler können verschiedene Bedingungen verursachen. Für die Kompilierzeit ist die häufigste Ursache ein falsch geschriebenes BASIC-Schlüsselwort oder -Argument. Während der Laufzeit wird er häufig durch eine unzulässig formatierte **DATA**-Anweisung verursacht. (Kompilierzeit- oder Laufzeitfehler)

ERR-Code: 2

`Syntaxfehler in numerischen Konstanten`

Eine numerische Konstante ist nicht richtig formatiert. (Kompilierzeitfehler)

`Typ besteht aus mehr als 65535 Bytes`

Ein benutzerdefinierter Typ darf 64K nicht überschreiten. (Kompilierzeitfehler)

`Typ nicht definiert`

Das Argument *Benutzertyp* der **TYPE**-Anweisung ist nicht definiert. (Kompilierzeitfehler)

`TYPE ohne END TYPE`

Eine **TYPE**-Anweisung hat keine zugehörige **END TYPE**-Anweisung. (Kompilierzeitfehler)

`TYPE-Anweisung falsch verschachtelt`

Definitionen von benutzerdefinierten Typen sind in Prozeduren nicht zulässig. (Kompilierzeitfehler)

`Überlauf`

Das Ergebnis einer Berechnung ist zu groß, um innerhalb des für Gleitkommazahlen oder Ganzzahlen erlaubten Bereiches dargestellt zu werden. (Laufzeitfehler)

ERR-Code: 6

`Überlauf des Datenspeichers`

Die Programmdaten sind zu umfangreich und passen nicht in den Speicher. Dieser Fehler wird häufig durch zu viele Konstanten oder zu viele statische Datenfeld-Elemente verursacht. Wenn Sie den Befehl BC oder die Optionen **EXE-Datei erstellen** oder **Bibliothek erstellen** verwenden, versuchen Sie, Debug-Optionen auszuschalten. Wenn der Speicher immer noch zu voll ist, teilen Sie das Programm auf und verwenden Sie die **CHAIN**-Anweisung oder den **$DYNAMIC**-Metabefehl. (Kompilierzeitfehler)

`Überlauf des Kommunikationspuffers`

Während einer Datenfernübertragung ist der Empfangspuffer übergelaufen. Die Größe des Empfangspuffers wird durch die Befehlszeilenoption /C oder durch die Option RB in der **OPEN COM**-Anweisung gesetzt. Versuchen Sie, den Puffer häufiger zu prüfen (mit der **LOC**-Funktion) oder ihn häufiger zu leeren (mit der **INPUT$**-Funktion). (Laufzeitfehler)

ERR-Code: 69

`Überlauf des Programmspeichers`

Sie versuchen, ein Programm zu kompilieren, dessen Codesegment größer als 64K ist. Versuchen Sie, das Programm in einzelne Module aufzuteilen, oder verwenden Sie die **CHAIN**-Anweisung. (Kompilierzeitfehler)

`Überlauf in numerischen Konstanten`

Die numerische Konstante ist zu groß. (Kompilierzeitfehler)

`Überspringen bis END TYPE-Anweisung`

Ein Fehler in der **TYPE**-Anweisung hat QuickBASIC veranlaßt, alle Daten zwischen der **TYPE**- und **END TYPE**-Anweisung zu ignorieren. (Kompilierzeitfehler)

`Umbenennen zwischen Disketten/Festplatte`

Ein Versuch, eine Datei mit einer neuen Laufwerkskennung umzubenennen. Das ist nicht erlaubt. (Laufzeit-Anfrage)

ERR-Code: 74

`Unbekannte Anweisung`

Sie haben wahrscheinlich eine BASIC-Anweisung falsch geschrieben. (Kompilierzeitfehler)

`Unbekannte Option:` *Option*

Sie haben eine unzulässige Option angegeben. (BC-Aufruffehler)

`Unerwartetes Dateiende in TYPE-Deklaration`

Innerhalb eines **TYPE...END TYPE**-Blockes befindet sich ein Dateiendezeichen.

`Ungültige Konstante`

Ein ungültiger Ausdruck wird benutzt, um einer Konstanten einen Wert zuzuweisen. Es ist zu beachten, daß Konstanten zugewiesene Ausdrücke, numerische Konstanten, symbolische Konstanten, jeden arithmetischen oder logischen Operator außer Potenzierung enthalten dürfen. Ein Zeichenkettenausdruck, der einer Konstanten zugewiesen wird, darf nur aus einer einzigen literalen Zeichenkette bestehen. (Kompilierzeitfehler)

`Ungültiges DECLARE für BASIC-Prozedur`

Sie versuchen, die Schlüsselworte **ALIAS**, **CDECL** oder **BYVAL** der **DECLARE**-Anweisung für eine BASIC-Prozedur zu benutzen. **ALIAS**, **CDECL** oder **BYVAL** können nur mit Nicht-BASIC-Prozeduren verwendet werden. (Kompilierzeitfehler)

`Ungültiges Zeichen`

QuickBASIC hat in der Quelldatei ein unzulässiges Zeichen, z. B. ein Steuerzeichen, gefunden. (Kompilierzeitfehler)

`Untere Grenze überschreitet obere Grenze`

Die untere Grenze überschreitet die in einer **DIM**-Anweisung definierte obere Grenze. (Kompilierzeitfehler)

`Unterprogramm nicht definiert`

Ein aufgerufenes Unterprogramm ist nicht definiert. (Kompilierzeitfehler)

`Unterprogramm unzulässig in Steueranweisungen`

FUNCTION-Definitionen der Unterprogramme sind in steuernden Konstruktionen wie **IF...THEN...ELSE** und **SELECT CASE** unzulässig. (Kompilierzeitfehler)

`Unterprogrammfehler`

Dieser Fehler tritt meistens bei **SUB**- oder **FUNCTION**-Definitionen auf und kann eine der folgenden Ursachen haben:

- **SUB** oder **FUNCTION** ist bereits definiert.
- Das Programm enthält falsch verschachtelte **SUB**- oder **FUNCTION**-Anweisungen.
- **SUB** oder **FUNCTION** wird nicht mit einer **END SUB**- oder **END FUNCTION**-Anweisung beendet.

(Kompilierzeitfehler)

`Unverträgliche Anzahl an Argumenten`

Sie verwenden eine falsche Argumentenanzahl mit einem BASIC-Unterprogramm oder einer BASIC-Funktion. (Kompilierzeitfehler)

`Unverträgliche Datentypen`

Die Variable ist nicht vom geforderten Typ. Sie versuchen beispielsweise, die **SWAP**-Anweisung mit einer Zeichenkettenvariablen und einer numerischen Variablen auszuführen. (Kompilierzeit- oder Laufzeitfehler)

ERR-Code: 13

`Unverträgliche Parametertypen`

Der Typ eines Unterprogramm- oder **FUNCTION**-Parameters entspricht nicht dem **DECLARE**-Anweisungs-Argument oder dem aufrufenden Argument. (Kompilierzeitfehler)

`Unzulässig außerhalb eines TYPE-Blockes`

Die Klausel *Element* **AS** *Typ* ist nur innerhalb eines **TYPE...END TYPE**-Blockes erlaubt. (Kompilierzeitfehler)

`Unzulässig außerhalb von SUB, FUNCTION oder DEF FN`

Diese Anweisung ist im Modul-Ebenen-Code nicht erlaubt. (Kompilierzeitfehler)

`Unzulässig außerhalb von SUB/FUNCTION`

Die Anweisung ist im Modul-Ebenen-Code oder in **DEF FN**-Funktionen nicht zulässig. (Kompilierzeitfehler)

`Unzulässig im Direktmodus`

Diese Anweisung ist nur innerhalb eines Programmes zulässig und kann im Direkt-Fenster nicht verwendet werden. (Kompilierzeitfehler)

`Unzulässige FOR-Indexvariable`

Dieser Fehler wird normalerweise verursacht, wenn ein falscher Variablentyp in einem **FOR**-Schleifenindex verwendet wird. Eine **FOR**-Schleifen-Indexvariable muß eine einfache numerische Variable sein. (Kompilierzeitfehler)

`Unzulässige Indexsyntax`

Ein Datenfeld-Index enthält einen Syntaxfehler, wie z. B. sowohl Zeichenketten- als auch ganzzahlige Datentypen. (Kompilierzeitfehler)

`Unzulässige Zahl`

Das Format der Zahl entspricht keinem gültigen Zahlenformat. Grund für die Fehlermeldung war wahrscheinlich ein Tippfehler, wie z.B. die Zahl `2p3`. (Kompilierzeitfehler)

`Unzulässiger COMMON-Name`

QuickBASIC hat eine unzulässige */Blockname/*-Angabe (z. B. ein *Blockname*, der ein BASIC-reserviertes Wort ist) in einem benannten **COMMON** entdeckt. (Kompilierzeitfehler)

`Unzulässiger Dateiname`

Bei **LOAD, SAVE, KILL** oder **OPEN** wird ein unzulässiges Format für den Dateinamen verwendet (z. B. ein Name mit zu vielen Zeichen). (Laufzeitfehler)

ERR-Code: 64

`Unzulässiger Funktionsaufruf`

Einer mathematischen oder Zeichenkettenfunktion wird ein Parameter übergeben, der außerhalb des Bereiches liegt. Außerdem kann ein Funktionsaufruffehler folgende Gründe haben:

- Es wird ein negativer oder übermäßig großer Index verwendet.
- Eine negative Zahl wird mit einer Zahl potenziert, die keine Ganzzahl ist.
- Bei der Verwendung von **GET** *Datei* oder **PUT** *Datei* wird eine negative Datensatznummer angegeben.
- Ein ungültiges oder bereichsüberschreitendes Argument wird an eine Funktion übergeben.
- Eine **BLOAD**- oder **BSAVE**-Operation wird in ein Nicht-Diskettengerät geleitet.
- Eine E/A-Funktion oder -Anweisung (z. B. **LOC** oder **LOF**) wird für ein Gerät ausgeführt, das diese nicht unterstützt.
- Zeichenketten werden verkettet und bilden eine Zeichenkette, die länger als 32.767 Zeichen ist.

(Laufzeitfehler)

ERR-Code: 5

`Unzulässiger Funktionsname`

Ein in BASIC reserviertes Wort wird als Name einer benutzerdefinierten **FUNCTION** verwendet. (Kompilierzeitfehler)

`Unzulässiges Trennzeichen`

In einer **PRINT USING**- oder **WRITE**-Anweisung steht ein unzulässiges Trennzeichen. Verwenden Sie ein Semikolon oder ein Komma. (Kompilierzeitfehler)

`Unzulässiges Typzeichen in einer numerischen Konstanten`

Eine numerische Konstante enthält ein unpassendes Typdeklarationszeichen. (Kompilierzeitfehler)

`Variable erforderlich`

QuickBASIC hat eine Anweisung **INPUT, LET, READ** oder **SHARED** ohne ein Variablenargument entdeckt. (Kompilierzeitfehler)

`Variable erforderlich`

Eine **GET**- oder **PUT**-Anweisung hat bei der Bearbeitung einer im **BINARY**-Modus geöffneten Datei keine Variable angegeben. (Laufzeitfehler)

ERR-Code: 40

`Variablenname bereits vorhanden`

Sie versuchen, *x* als benutzerdefinierten Typ zu definieren, nachdem bereits *x.y* verwendet wird. (Kompilierzeitfehler)

`Verschachtelte Funktionsdefinition`

Eine **FUNCTION**-Definition erscheint in einer anderen **FUNCTION**-Definition oder innerhalb einer **IF...THEN...ELSE**-Klausel. (Kompilierzeitfehler)

`WEND ohne WHILE`

Dieser Fehler wird verursacht, wenn eine **WEND**-Anweisung keine entsprechende **WHILE**-Anweisung hat. (Kompilierzeitfehler)

`WHILE ohne WEND`

Dieser Fehler wird verursacht, wenn eine **WHILE**-Anweisung keine entsprechende **WEND**-Anweisung hat. (Kompilierzeitfehler)

`Zeichenkette variabler Länge erforderlich`

In einer **FIELD**-Anweisung sind nur Zeichenketten variabler Länge erlaubt. (Kompilierzeitfehler)

`Zeichenketten fester Länge unzulässig`

Sie versuchen, eine Zeichenkette fester Länge als formalen Parameter zu benutzen. (Kompilierzeitfehler)

`Zeichenkettenausdruck erforderlich`

Die Anweisung erfordert als Argument einen Zeichenkettenausdruck. (Kompilierzeitfehler)

`Zeichenkettenbereich beschädigt`

Dieser Fehler tritt auf, wenn während einer Heap-Komprimierung aus dem Zeichenkettenbereich eine ungültige Zeichenkette gelöscht wird. Der Fehler tritt wahrscheinlich aus folgenden Gründen auf:

- Ein Zeichenkettenbeschreiber oder Zeichenkettenrückzeiger ist unzulässig verändert worden. Dieser Fehler kann eintreten, wenn Sie eine Assembler-Unterroutine zum Modifizieren von Zeichenketten verwenden.
- Außerhalb des Bereichs liegende Datenfeldindizes werden verwendet und der Zeichenkettenbereich wird unabsichtlich verändert. Die Option **Debug-Code erzeugen** kann während der Kompilierzeit verwendet werden, um nach Datenfeldindizes zu suchen, die die Datenfeldgrenzen überschreiten.
- Der falsche Gebrauch der Anweisungen **POKE** und/oder **DEF SEG** kann den Zeichenkettenbereich unzulässig verändern.
- Zwischen zwei verketteten Programmen können nicht-übereinstimmende **COMMON**-Deklarationen auftreten.

(Laufzeitfehler)

`Zeichenkettenformel zu umfangreich`

Eine Zeichenkettenformel ist zu lang, oder eine **INPUT**-Anweisung erfordert mehr als 15 Zeichenkettenvariablen. Teilen Sie die Formel oder die **INPUT**-Anweisung zur korrekten Ausführung auf. (Laufzeitfehler)

ERR-Code: 16

`Zeichenkettenvariable erforderlich`

Die Anweisung erfordert als Argument eine Zeichenkettenvariable. (Kompilierzeitfehler)

`Zeile ungültig. Erneut starten.`

Es wird ein ungültiges Dateinamen-Zeichen nach den Pfadzeichen "\" (umgekehrter Schrägstrich) oder ":" (Doppelpunkt) benutzt. (BC-Aufruffehler)

`Zeile zu lang`

Zeilenlängen sind auf 255 Zeichen begrenzt. (Kompilierzeitfehler)

`Zu viele Argumente im Funktionsaufruf`

Funktionsaufrufe sind auf 60 Argumente begrenzt. (Kompilierzeitfehler)

`Zu viele benannte COMMON-Blöcke`

Die erlaubte Höchstanzahl benannter **COMMON**-Blöcke beträgt 126. (Kompilierzeitfehler)

`Zu viele Dateien`

Während der Kompilierzeit tritt dieser Fehler auf, wenn Include-Dateien über mehr als fünf Ebenen verschachtelt sind. Er tritt während der Laufzeit auf, wenn das Verzeichnis-Maximum von 255 Dateien bei dem Versuch, eine neue Datei mit einer **SAVE**- oder **OPEN**-Anweisung anzulegen, überschritten wird. (Kompilierzeit- oder Laufzeitfehler)

ERR-Code: 67

`Zu viele Dimensionen`

Datenfelder sind auf 60 Dimensionen begrenzt. (Kompilierzeitfehler)

`Zu viele Marken`

Die Zeilenanzahl in der Zeilenliste, die der **ON...GOTO**- oder **ON...GOSUB**-Anweisung folgt, überschreitet 255, (Kompilierzeitfehler) oder 59 (Laufzeitfehler in kompilierten Anwendungen)

`Zu viele TYPE-Definitionen`

Die erlaubte Höchstanzahl von benutzerdefinierten Typen beträgt 240. (Kompilierzeitfehler)

`Zu viele Variablen für INPUT`

Eine **INPUT**-Anweisung ist auf 60 Variablen begrenzt. (Kompilierzeitfehler)

`Zu viele Variablen für LINE INPUT`

In einer **LINE INPUT**-Anweisung ist nur eine Variable erlaubt. (Kompilierzeitfehler)

`Zugriff nicht gestattet`

Es wurde versucht, auf eine schreibgeschützte Diskette zu schreiben oder auf eine gesperrte Datei zuzugreifen. (Laufzeitfehler)

ERR-Code: 70

`Zusätzlicher Dateiname nicht beachtet`

In der Befehlszeile sind zu viele Dateinamen angegeben, der letzte Dateiname wurde nicht beachtet. (BC-Aufruffehler)

`Zuweisung einer Zeichenkette erforderlich`

Bei einer **RSET**-Anweisung fehlt die Zuweisung der Zeichenkette. (Kompilierzeitfehler)

I.3 LINK-Fehlermeldungen

Dieser Abschnitt stellt die vom Microsoft Overlay Linker, LINK, erzeugten Fehlermeldungen vor.

Der Linker unterbricht die Ausführung beim Auftreten fataler Fehler, die folgendes Format haben:

Dateilage : `fataler Fehler L1`*xxx*`:` *Meldungstext*

Nicht fatale Fehler deuten auf Probleme in der von LINK erzeugten ausführbaren Datei hin und haben folgendes Format:

Dateilage : `Fehler L2`*xxx*`:` *Meldungstext*

Warnungen weisen auf mögliche Probleme in der von LINK erzeugten ausführbaren Datei hin und haben folgendes Format:

Dateilage : `Warnung L4`*xxx*`:` *Meldungstext*

In diesen Meldungen ist *Dateilage* die mit dem Fehler verknüpfte Eingabedatei oder LINK, falls keine Eingabedatei vorhanden ist.

Beim Binden von Objektdateien mit LINK können folgende Fehlermeldungen erscheinen:

Nummer ***LINK-Fehlermeldung***

`L1001` *Option* : `Optionsname nicht eindeutig`

Es erscheint kein eindeutiger Optionsname nach dem Optionsanzeiger (/). Dieser Fehler wird z.B. durch den Befehl

```
LINK /N main;
```

erzeugt, da LINK nicht erkennen kann, welche der drei Optionen, die mit "N" beginnen, gemeint ist.

`L1002` *Option* : `Unbekannter Optionsname`

Ein unbekanntes Zeichen folgt dem Optionsanzeiger (/), wie im folgenden Beispiel:

```
LINK /ABCDEF main;
```

`L1003` `/QUICKLIB, /EXEPACK nicht kompatibel`

Sie haben zwei Optionen angegeben, /QUICKLIB und /EXEPACK, die nicht zusammen verwendet werden können.

`L1004` *Option* : `Ungültiger numerischer Wert`

Für eine der Linker-Optionen erscheint ein falscher Wert. Zum Beispiel ist eine Zeichenkette für eine Option angegeben, die jedoch einen numerischen Wert erfordert.

`L1006` *Option* : `Stapelgröße überschreitet 65535 Bytes`

Der Wert, der als Parameter der Option /STACKSIZE übergeben wurde, überschreitet den erlaubten Höchstwert.

`L1007` *Option* : `Interrupt-Nummer überschreitet 255`

Eine Zahl größer als 255 wurde als Wert für die Option /OVERLAYINTERRUPT angegeben.

`L1008` *Option* : `Segmentlimit zu groß`

Beim Verwenden der Option /SEGMENTS ist die erlaubte Höchstanzahl an Segmenten größer als 3072 gesetzt.

`L1009` *Nummer* : `CPARMAXALLOC: Unzulässsiger Wert`

Die Zahl, die in der Option /CPARMAXALLOC angegeben wird, befindet sich nicht im Bereich von 1-65.535.

`L1020` `Objektmodule nicht festgelegt`

Dem Linker wurden keine Objektdateinamen angegeben.

`L1021` `Antwortdateien können nicht verschachtelt werden`

Eine Antwortdatei (Response File) erscheint innerhalb einer Antwortdatei.

`L1022` `Antwortzeile zu lang`

Eine Zeile in einer Antwortdatei ist länger als 127 Zeichen.

`L1023` `Vom Benutzer abgebrochen`

Sie haben STRG+C oder STRG+UNTBR eingegeben.

`L1024` `Verschachtelte rechte Klammern`

Der Inhalt eines Overlays wurde auf der Befehlszeile falsch eingegeben.

`L1025` `Verschachtelte linke Klammern`

Der Inhalt eines Overlays wurde auf der Befehlszeile falsch eingegeben.

L1026 `Fehlende rechte Klammer`

Eine rechte Klammer fehlt in der Inhaltsbeschreibung eines Overlays auf der Befehlszeile.

L1027 `Fehlende linke Klammer`

Eine linke Klammer fehlt in der Inhaltsbeschreibung eines Overlays auf der Befehlszeile.

L1043 `Überlauf der Verschiebungstabelle`

Es erscheinen mehr als 32.768 lange Aufrufe, lange Sprünge oder andere lange Zeiger in dem Programm.

Versuchen Sie, lange Verweise nach Möglichkeit durch kurze Verweise zu ersetzen, und erstellen Sie das Objektmodul erneut.

L1045 `Zu viele TYPDEF-Datensätze`

Ein Objektmodul enthält mehr als 255 TYPDEF-Datensätze. Diese Datensätze beschreiben gemeinsame Variablen. Dieser Fehler kann nur bei Programmen auftreten, die mit dem Microsoft FORTRAN-Compiler oder anderen Compilern erstellt wurden, die gemeinsame Variablen unterstützen. (TYPDEF ist ein DOS-Ausdruck. Er wird im *Microsoft MS-DOS Programmer's Reference* und anderen DOS-Nachschlagewerken erklärt.)

L1046 `Zu viele externe Symbole in einem Modul`

Ein Objektmodul überschreitet die Höchstgrenze von 1023 externen Symbolen.

Teilen Sie das Modul in kleinere Abschnitte auf.

L1047 `Zu viele Gruppen-, Segment- und Klassennamen in einem Modul`

Das Programm enthält zuviele Gruppen-, Segment- und Klassennamen.

Verringern Sie die Anzahl der Gruppen, Segmente und Klassen, und erstellen Sie die Objektdatei erneut.

L1048 `Zu viele Segmente in einem Modul`

Ein Objektmodul hat mehr als 255 Segmente.

Teilen Sie das Modul auf oder fügen Sie Segmente zusammen.

L1049 `Zu viele Segmente`

Das Programm enthält mehr als die Höchstanzahl von Segmenten. Verwenden Sie die Option /SEGMENTS, die die zulässige Höchstanzahl von Segmenten standardmäßig auf 128 festlegt.

Binden Sie erneut unter Verwendung der Option /SEGMENTS mit einer passenden Anzahl von Segmenten.

`L1050` `Zu viele Gruppen in einem Modul`

LINK stellte mehr als 21 Gruppen-Definitionen (GRPDEF) in einem einzigen Modul fest.

Verringern Sie die Anzahl der Gruppen-Definitionen oder teilen Sie das Modul auf. (Gruppen-Definitionen werden in *Microsoft MS-DOS Programmer's Reference* und in anderen Nachschlagewerken zu DOS erklärt.)

`L1051` `Zu viele Gruppen`

Das Programm definiert mehr als 20 Gruppen, DGROUP nicht mitgezählt.

Verringern Sie die Anzahl der Gruppen.

`L1052` `Zu viele Bibliotheken`

Erscheint beim Versuch, mit mehr als 32 Bibliotheken zu binden.

Kombinieren Sie Bibliotheken, oder verwenden Sie Module, die weniger Bibliotheken erfordern.

`L1053` `Symboltabellen-Überlauf`

Es gibt keine feste Obergrenze für die Größe der Symboltabelle. Diese ist jedoch durch die Größe des verfügbaren Speicherplatzes beschränkt.

Kombinieren Sie Module oder Segmente und erstellen Sie die Objektdatei erneut. Entfernen Sie so viele globale Symbole wie möglich.

`L1054` `Erfordertes Segmentlimit zu hoch`

LINK hat nicht genügend Speicher, um Tabellen zuzuweisen, die die Anzahl der erforderten Segmente beschreiben. (Die Standardeinstellung ist 128 oder der mit der Option /SEGMENTS festgelegte Wert.)

Versuchen Sie unter Verwendung der Option /SEGMENTS erneut zu binden, um eine kleinere Anzahl von Segmenten auszuwählen (benutzen Sie z. B. 64, wenn Sie zunächst die Standardeinstellung benutzt haben) oder schaffen Sie Speicherplatz durch Entfernen speicherresidenter Programme oder Shells.

`L1056` `Zu viele Overlays`

Das Programm definiert mehr als 63 Overlays.

`L1057` `Datensatz zu lang`

Ein LEDATA-Satz (in einem Objektmodul) enthält mehr als 1024 Byte Daten. Es handelt sich um einen Übersetzerfehler. (LEDATA ist ein DOS-Ausdruck, der im *Microsoft MS-DOS Programmer's Reference* und in anderen DOS-Nachschlagewerken erklärt wird.)

L1063 `Ungenügend Speicherplatz für CodeView Information`

Zu viele Objektdateien ("*.obj*") enthalten Debug-Informationen. Schalten Sie die Option **Debug-Code erzeugen** im Dialogfeld **EXE-Datei erstellen** aus.

L1070 `Segmentgröße überschreitet 64K`

Ein einzelnes Segment enthält mehr als 64K Code oder Daten.

Versuchen Sie, unter Verwendung des Large-Modells zu kompilieren und zu binden.

L1071 `Segment _TEXT größer als 65520 Bytes`

Dieser Fehler tritt wahrscheinlich nur in C-Programmen vom Typ Small-Modell auf. Er kann jedoch auch auftreten, wenn Programme mit einem Segment namens _TEXT, unter Verwendung der Option /DOSSEG, gebunden werden. C-Programme vom Typ Small-Modell müssen die Code-Adressen 0 und 1 reservieren; dieser Bereich wird für Ausrichtungszwecke auf 16 erhöht.

L1072 `Common-Bereich größer als 65536 Bytes`

Das Programm enthält mehr als 64K gemeinsamer Variablen. Dieser Fehler tritt nur bei Programmen auf, die von Compilern erstellt wurden, die gemeinsame Variablen unterstützen.

L1080 `List-Datei kann nicht geöffnet werden`

Die Diskette/Festplatte oder das Hauptverzeichnis ist voll.

Löschen oder entfernen Sie Dateien, um Speicherplatz zu schaffen.

L1081 `Kein Speicherplatz für die zu startende Datei vorhanden`

Die Diskette/Festplatte, auf die die *.exe*-Datei geschrieben werden soll, ist voll.

Schaffen Sie Speicherplatz auf der Diskette/Festplatte und starten Sie den Linker erneut.

L1083 `Zu startende Datei kann nicht geöffnet werden`

Die Diskette/Festplatte oder das Hauptverzeichnis ist voll.

Löschen oder entfernen Sie Dateien, um Speicherplatz zu schaffen.

L1084 `Temporäre Datei kann nicht erstellt werden`

Die Diskette/Festplatte oder das Hauptverzeichnis ist voll.

Schaffen Sie Platz in dem Verzeichnis und starten Sie den Linker erneut.

L1085 `Temporäre Datei kann nicht geöffnet werden`

Die Diskette/Festplatte oder das Hauptverzeichnis ist voll.

Löschen oder entfernen Sie Dateien, um Speicherplatz zu schaffen.

L1086 `Temporäre Datei fehlt`

Ein interner Fehler ist aufgetreten.

Teilen Sie Microsoft die Bedingungen, unter denen dieser Fehler aufgetreten ist, im beiliegenden Formblatt mit.

L1087 `Unerwartetes Dateiende auf temporärer Datei`

Die Diskette mit der temporären Linker-Ausgabedatei wurde entfernt.

L1088 `Kein Speicherplatz für List-Datei vorhanden`

Die Diskette/Festplatte, auf die die Listdatei geschrieben wird, ist voll.

Schaffen Sie Speicherplatz auf der Diskette/Festplatte und starten Sie den Linker erneut.

L1089 *Dateiname* : `Antwortdatei kann nicht geöffnet werden`

LINK konnte die angegebene Antwortdatei nicht finden.

Es handelt sich normalerweise um einen Schreibfehler.

L1090 `List-Datei kann nicht erneut geöffnet werden`

Die ursprüngliche Diskette wurde bei der Anfrage nicht wieder eingelegt.

Starten Sie den Linker erneut.

L1091 `Unerwartetes Dateiende in Bibliothek`

Die Diskette, die die Bibliothek enthält, ist wahrscheinlich entfernt worden.

Legen Sie die Diskette wieder ein, und starten Sie den Linker erneut.

L1093 `Objekt nicht gefunden`

Eine der in der Linker-Eingabe angegebenen Objektdateien wurde nicht gefunden.

Starten Sie den Linker erneut und geben Sie die Objektdatei an.

L1101 `Ungültiges Objektmodul`

Eines der Objektmodule ist ungültig.

Wenn der Fehler nach nochmaligem Kompilieren auftritt, teilen Sie dies bitte Microsoft in beiliegendem Formblatt mit.

L1102 `Unerwartetes Dateiende`

Es tritt ein ungültiges Format für eine Bibliothek auf.

L1103 `Versuch, auf Daten außerhalb der Segmentgrenzen zuzugreifen`

Ein Datensatz in einem Objektmodul gibt Daten an, die über das Segmentende hinausgehen. Es handelt sich um einen Übersetzerfehler.

Stellen Sie den Übersetzer (Compiler oder Assembler), der das falsche Objektmodul erzeugt hat, und die entsprechenden Bedingungen fest. Bitte teilen Sie Microsoft diesen Fehler im beiliegenden Formblatt mit.

L1104 *Dateiname* `: Ungültige Bibliothek`

Die angegebene Datei ist keine gültige Bibliotheksdatei. Dieser Fehler veranlaßt LINK abzubrechen.

L1113 `Nicht aufgelöste COMDEF; interner Fehler`

Teilen Sie Microsoft die Bedingungen, unter denen dieser Fehler aufgetreten ist, im beiliegenden Formblatt mit.

L1114 `Datei nicht passend zu /EXEPACK; binden Sie erneut ohne`

Für das gebundene Programm ist das gepackte Ladebild plus Pack-Verwaltung größer als das ungepackte Ladebild.

Binden Sie erneut ohne die Option /EXEPACK.

L1115 `/QUICKLIB, Overlays nicht kompatibel`

Sie haben Overlays spezifiziert und verwenden die Option /QUICKLIB. Sie können jedoch nicht beide zusammen verwenden.

L2001 `Fixup(s) ohne Daten`

Ein FIXUPP-Datensatz kommt ohne einen ihm direkt vorhergehenden Datensatz vor. Es handelt sich wahrscheinlich um einen Compilerfehler. (Im *Microsoft MS-DOS Programmer's Reference* finden Sie weitere Informationen zu FIXUPP.)

L2002 `Fixup-Überlauf nahe` *Zahl* `im Segment` *Segname*

Folgende Bedingungen können diesen Fehler verursachen:

- Eine Gruppe ist größer als 64K.
- Das Programm enthält einen kurzen Intersegment-Sprung oder kurzen Intersegment-Aufruf.
- Der Name eines Datenobjekts in dem Programm kollidiert mit dem einer Bibliotheksunterroutine, die in den Linkprozess einbezogen ist.
- Eine **EXTRN**-Deklaration in einer Assembler-Quelldatei erscheint innerhalb eines Segments, wie im folgenden Beispiel:

```
code   SEGMENT public 'CODE'
       EXTRN    main:far
start  PROC     far
       call     main
       ret
start  ENDP
code   ENDS
```

Die folgende Konstruktion ist zu bevorzugen:

```
EXTRN   main:far
code    SEGMENT public 'CODE'
start   PROC    far
        call    main
        ret
start   ENDP
code    ENDS
```

Überprüfen Sie die Quelldatei und erstellen Sie die Objektdatei erneut. (Weitere Informationen zu Rahmen- und Ziel-Segmenten finden Sie im *Microsoft MS-DOS Programmer's Reference.)*

L2003 `Selbstbezogener Intersegment-Fixup bei` *Offset* `im Segment` *Segname*

Erscheint beim Versuch, einen kurzen Aufruf oder Sprung zu einem langen Eintrag im Segment *Segname* bei *Offset* durchzuführen.

Ändern Sie den Aufruf oder Sprung in einen langen Aufruf oder Sprung, oder machen Sie den Eintrag kurz.

L2004 `Fixup-Überlauf vom Typ LOBYTE`

Ein Fixup vom Typ LOBYTE hat einen Adressen-Überlauf verursacht. Nähere Informationen sind *Microsoft MS-DOS Programmer's Reference* zu entnehmen.

L2005 `Fixup-Typ nicht unterstützt`

Ein Fixup-Typ tritt auf, der nicht von dem Microsoft-Linker unterstützt wird. Es handelt sich wahrscheinlich um einen Compilerfehler.

Teilen Sie bitte Microsoft die Bedingungen, unter denen dieser Fehler aufgetreten ist, im beiliegenden Formblatt mit.

L2011 *Name* `: NEAR/HUGE-Konflikt`

Einer gemeinsamen Variablen sind kollidierende NEAR- und HUGE-Attribute zugewiesen worden. Dieser Fehler kann nur bei Programmen auftreten, die mit Compilern erstellt wurden, die gemeinsame Variablen unterstützen.

L2012 *Name* : `Unverträgliche Datenfeldelementgröße`

Ein langes gemeinsames Datenfeld wird mit zwei oder mehr verschiedenen Datenfeldelementgrößen deklariert. (Zum Beispiel wurde ein Datenfeld einmal als Zeichendatenfeld und einmal als Datenfeld mit reellen Zahlen deklariert.) Dieser Fehler tritt nur bei Compilern auf, die lange gemeinsame Datenfelder unterstützen.

L2013 `LIDATA-Satz zu groß`

Ein LIDATA-Satz enthält mehr als 512 Bytes. Dieser Fehler wird normalerweise durch einen Compiler-Fehler verursacht.

L2024 *Name* : `Symbol bereits definiert`

Der Linker hat ein redefiniertes globales Symbol gefunden. Entfernen Sie die andere(n) Definition(en).

L2025 *Name* : `Symbol mehr als einmal definiert`

Entfernen Sie die doppelte Symboldefinition aus der Objektdatei.

L2029 `Nicht aufgelöste externe Symbole`

Ein oder mehrere Symbole sind in einem oder mehreren Modulen als extern deklariert, sind aber in keinem der Module oder keiner der Bibliotheken als global definiert. Eine Liste der nicht aufgelösten externen Verweise erscheint, wie im nachfolgenden Beispiel gezeigt, nach der Meldung.

```
Nicht aufgelöste externe Symbole
EXIT in Datei(en):
 HAUPT.OBJ (haupt.for)
OPEN in Datei(en):
 HAUPT.OBJ (haupt.for)
```

Der Name, der vor `in Datei(en)` steht, ist das nicht aufgelöste externe Symbol. Die nächste Zeile enthält eine Liste der Objektmodule, die Bezug auf dieses Symbol nehmen. Diese Meldung und die Liste werden ebenfalls in die Map-Datei geschrieben, sofern eine vorhanden ist.

L2041 `Stapel plus Daten überschreiten 64K`

Die Gesamtgröße von Stapel und Near-Daten überschreitet 64K. Reduzieren Sie die Größe des Stapels, um diesen Fehler zu beheben.

Der Linker testet diese Bedingung nur, wenn die Option /DOSSEG gesetzt wurde. Diese Option wird automatisch durch das Bibliotheksstartmodul gesetzt.

L2043 `Quick-Bibliothek-Einrichtungsmodul fehlt`

Sie haben das Objektmodul oder die Bibliothek zur Einrichtung einer Quick-Bibliothek nicht angegeben bzw. LINK kann diese nicht finden. Im Fall von QuickBASIC wird die Bibliothek durch *bqlb45.lib* bereitgestellt.

L2044 *Name* `: Symbol mehrfach definiert; /NOE verwenden`

Der Linker hat eine mögliche erneute Definition eines globalen Symbols gefunden. Dieser Fehler wird oft durch die erneute Definition eines in der Bibliothek definierten Symbols verursacht.

Binden Sie erneut unter Verwendung der Option /NOEXTDICTIONARY.

Dieser Fehler, zusammen mit dem Fehler `L2025` für dasselbe Symbol, zeigen einen realen Redefinitions-Fehler an.

L4011 `PACKCODE-Wert über 65500 ist unzuverlässig`

Packcode-Segmentgrößen, die 65.500 Bytes überschreiten, können beim Prozessor Intel 80286 unzuverlässig sein.

L4012 `HIGH Laden schaltet EXEPACK aus`

Die Optionen /HIGH und /EXEPACK können nicht gleichzeitig verwendet werden.

L4015 `/CODEVIEW schaltet /DSALLOCATE aus`

Die Optionen /CODEVIEW und /DSALLOCATE können nicht gleichzeitig verwendet werden.

L4016 `/CODEVIEW schaltet /EXEPACK aus`

Die Optionen /CODEVIEW und /EXEPACK können nicht gleichzeitig verwendet werden.

L4020 *Name* `: Codesegmentgröße überschreitet 65500`

Codesegmente von 65.501 bis 65.536 Bytes Länge können beim Intel 80286-Prozessor unzuverlässig sein.

L4021 `Kein Stapelsegment`

Das Programm enthält kein Stapelsegment, das mit dem Kombiniertyp **STACK** definiert ist. Diese Meldung dürfte für Module, die mit Microsoft QuickBASIC kompiliert sind, nicht erscheinen, kann aber bei einem Assembler-Modul auftreten.

Normalerweise sollte jedes Programm ein Stapelsegment mit dem Kombiniertyp **STACK** haben. Sie können diese Meldung ignorieren, wenn Sie einen besonderen Grund haben, keinen Stapel bzw. einen Stapel ohne den Kombinier-Typ **STACK** zu definieren. Das Binden mit Linker-Versionen kleiner als Version 2.40 kann diese Meldung verursachen, da diese Linker die Bibliotheken nur einmal durchsuchen.

L4031 *Name* : `Segment in mehr als einer Gruppe deklariert`

Ein Segment wurde als Teil zweier verschiedener Gruppen deklariert.

Korrigieren Sie die Quelldatei und erstellen Sie die Objektdateien erneut.

L4034 `Mehr als 239 Overlay-Segmente; Extras im Stamm`

Das Programm setzt mehr als 239 Segmente in die Overlays. Beim Auftreten dieses Fehlers werden die Segmente beginnend mit der Nummer 234 im permanent residenten Teil, dem Stamm, plaziert.

L4045 `Name der Ausgabedatei ist` *Name*

Die Anfrage (Prompt) für das Feld der ausführbaren Datei gibt einen fehlerhaften Standardwert an, da /QUICKLIB zu spät verwendet wurde. Die Ausgabe ist eine Quick-Bibliothek mit dem in der Fehlermeldung angegebenen Namen.

L4050 `Zu viele globale Symbole zu sortieren`

Die Anzahl der globalen Symbole überschreitet den Speicherplatz für die Sortierung der Symbole, der durch die /MAP-Option angefordert wurde. Die Symbole bleiben unsortiert.

L4051 *Dateiname* : `Kann Bibliothek nicht finden`

Der Linker kann die angegebene Datei nicht finden.

Geben Sie entweder einen neuen Dateinamen, eine neue Pfadbeschreibung oder beides ein.

L4053 `VM.TMP : unzulässiger Dateiname; ignoriert`

vm.tmp erscheint als Name einer Objektdatei. Benennen Sie die Datei um und starten Sie den Linker erneut.

L4054 *Dateiname* : `Kann Datei nicht finden`

Der Linker kann die angegebene Datei nicht finden. Geben Sie entweder einen neuen Dateinamen, eine neue Pfadbeschreibung oder beides ein.

I.4 LIB-Fehlermeldungen

Die vom Microsoft-Bibliotheksmanager LIB erzeugten Fehlermeldungen haben eines der folgenden Formate:

{*Dateiname* | `LIB`} : `fataler FehlerU1s`*xx : Text der Meldung*
{*Dateiname* | `LIB`} : `Fehler U2`*xxx : Text der Meldung*
{*Dateiname* | `LIB`} : `Warnung U4`*xxx : Text der Meldung*

Die Fehlermeldung beginnt mit dem Namen der Eingabedatei (*Dateiname*), sofern eine existiert, oder mit dem Namen des Dienstprogrammes. Wenn möglich, schreibt LIB eine Warnung und fährt mit der Ausführung fort. In einigen Fällen sind Fehler fatal und LIB beendet die Verarbeitung.

LIB kann die folgenden Fehlermeldungen anzeigen:

Nummer LIB-Fehlermeldung

`U1150` `Seitengröße zu klein`

Die Seite einer eingegebenen Bibliothek war zu klein, was auf eine ungültig eingegebene *.lib*-Datei hindeutet.

`U1151` `Syntaxfehler : Ungültige Dateiangabe`

Ein Befehlsoperator, wie z. B. ein Minuszeichen (-), wird ohne einen darauf folgenden Modulnamen angegeben.

`U1152` `Syntaxfehler : Fehlender Optionsname`

Ein Schrägstrich (/) wird ohne eine darauffolgende Option angegeben.

`U1153` `Syntaxfehler : Fehlender Optionswert`

Die Option /PAGESIZE wird ohne einen darauffolgenden Wert angegeben.

`U1154` `Option unbekannt`

Es wird eine unbekannte Option eingegeben. Üblicherweise erkennt LIB nur die Option /PAGESIZE.

`U1155` `Syntaxfehler : Unzulässige Eingabe`

Der eingegebene Befehl entspricht nicht der korrekten LIB-Syntax, wie diese im Anhang G, "Kompilieren und Binden aus DOS", angegeben ist.

`U1156` `Syntaxfehler`

Der eingegebene Befehl entspricht nicht der korrekten LIB-Syntax, wie diese im Anhang G, "Kompilieren und Binden aus DOS", angegeben ist.

`U1157` `Komma oder neue Zeile fehlt`

Ein Komma oder ein Wagenrücklauf wird in der Befehlszeile erwartet, erscheint jedoch nicht. Dies kann auf ein falsch plaziertes Komma hinweisen, wie in der folgenden Zeile:

```
LIB math.lib,-mod1+mod2;
```

Die Zeile hätte wie folgt eingegeben werden müssen:

```
LIB math.lib -mod1+mod2;
```

U1158 `Fehlendes Endezeichen`

Entweder die Antwort auf die Anfrage `"Ausgabe-Bibliothek"` oder die letzte Zeile der zum Start von LIB benutzten Antwortdatei wurde nicht mit einem Wagenrücklauf abgeschlossen.

U1161 `Alte Bibliothek kann nicht umbenannt werden`

LIB kann die alte Bibliothek mit der Erweiterung *.bak* nicht umbenennen, da die *.bak*-Version bereits mit einem Schreibschutz versehen ist.

Ändern Sie den Schutz der alten *.bak*-Version.

U1162 `Bibliothek kann nicht erneut geöffnet werden`

Die alte Bibliothek kann nach Umbenennen mit einer *.bak*-Erweiterung nicht neu geöffnet werden.

U1163 `Fehler beim Schreiben in Datei mit externen Querverweisen`

Diskette/Festplatte oder Hauptverzeichnis ist voll.

Löschen oder entfernen Sie Dateien, um Speicherplatz zu schaffen.

U1170 `Zu viele Symbole`

Es erscheinen mehr als 4609 Symbole in der Bibliotheksdatei.

U1171 `Unzureichender Speicherplatz`

LIB hat nicht genug Speicher, um zu starten.

Beseitigen Sie alle Shells oder speicherresidenten Programme und versuchen Sie erneut zu starten, oder fügen Sie mehr Speicherplatz hinzu.

U1172 `Kein virtueller Speicher mehr vorhanden`

Teilen Sie Microsoft die Bedingungen, unter denen der Fehler aufgetreten ist, im beiliegenden Formblatt mit.

U1173 `Interner Fehler`

Teilen Sie Microsoft die Bedingungen, unter denen der Fehler aufgetreten ist, im beiliegenden Formblatt mit.

U1174 `Kennzeichen : Nicht zugewiesen`

Teilen Sie Microsoft die Bedingungen, unter denen der Fehler aufgetreten ist, im beiliegenden Formblatt mit.

U1175 `Frei : Nicht zugewiesen`

Teilen Sie Microsoft die Bedingungen, unter denen der Fehler aufgetreten ist, im beiliegenden Formblatt mit.

U1180 `Schreiben in Auszugsdatei nicht möglich`

Die Diskette/Festplatte oder das Hauptverzeichnis ist voll.

Löschen oder entfernen Sie Dateien, um Speicherplatz zu schaffen.

U1181 `Schreiben in Bibliotheksdatei nicht möglich`

Die Diskette/Festplatte oder das Hauptverzeichnis ist voll.

Löschen oder entfernen Sie Dateien, um Speicherplatz zu schaffen.

U1182 *Dateiname* : `Auszugsdatei kann nicht angelegt werden`

Die Diskette/Festplatte oder das Hauptverzeichnis ist voll oder die angegebene Auszugsdatei existiert bereits mit einem Schreibschutz.

Schaffen Sie Speicherplatz auf der Diskette/Festplatte oder ändern Sie den Schutz der Auszugsdatei.

U1183 `Antwortdatei kann nicht geöffnet werden`

Die Antwortdatei wurde nicht gefunden.

U1184 `Unerwartetes Dateiende bei Befehlseingabe`

Ein Dateiendezeichen wurde zu früh als Antwort auf eine Anfrage empfangen.

U1185 `Neue Bibliothek kann nicht erstellt werden`

Die Diskette/Festplatte oder das Hauptverzeichnis ist voll oder die Bibliotheksdatei existiert bereits mit einem Schreibschutz.

Schaffen Sie Platz auf der Diskette/Festplatte oder in dem Verzeichnis oder entfernen Sie den Schreibschutz.

U1186 `Fehler beim Schreiben in neue Bibliothek`

Die Diskette/Festplatte oder das Hauptverzeichnis ist voll.

Löschen oder entfernen Sie Dateien, um Speicherplatz zu schaffen.

U1187 `VM.TMP kann nicht geöffnet werden`

Die Diskette/Festplatte oder das Hauptverzeichnis ist voll.

Löschen oder entfernen Sie Dateien, um Speicherplatz zu schaffen.

U1188 `In VM kann nicht geschrieben werden`

Teilen Sie Microsoft die Bedingungen, unter denen der Fehler aufgetreten ist, im beiliegenden Formblatt mit.

U1189 `Aus VM kann nicht gelesen werden`

Teilen Sie Microsoft die Bedingungen, unter denen der Fehler aufgetreten ist, im beiliegenden Formblatt mit.

U1190 `Vom Benutzer unterbrochen`

Der Benutzer hat STRG+C oder STRG+UNTBR betätigt.

U1200 *Name* : `Ungültiger Bibliothekskopf`

Die eingegebene Bibliotheksdatei hat ein ungültiges Format. Sie ist entweder keine Bibliotheksdatei, oder sie ist fehlerhaft.

U1203 *Name* : `Ungültiges Objektmodul nahe Dateilage`

Das mit *Name* gekennzeichnete Modul ist kein gültiges Objektmodul.

U2152 *Dateiname* : `Listing kann nicht erstellt werden`

Das Verzeichnis oder die Diskette/Festplatte ist voll, oder die Listingdatei der Querverweise existiert bereits mit Schreibschutz.

Schaffen Sie Platz auf der Diskette/Festplatte oder ändern Sie den Schutz der Listingdatei der Querverweise.

U2155 *Modulname* : `Modul nicht in Bibliothek; ignoriert`

Das angegebene Modul wurde nicht in der eingegebenen Bibliothek gefunden.

U2157 *Dateiname* : `Auf Datei kann nicht zugegriffen werden`

LIB kann die angegebene Datei nicht öffnen.

U2158 *Bibliotheksname* : `Ungültiger Bibliothekskopf; Datei ignoriert`

Die eingegebene Bibliothek hat ein ungültiges Format.

U2159 *Dateiname* : `Ungültiges Format Hexzahl; Datei ignoriert`

Das Signatur-Byte oder -Wort *Hexzahl* der gegebenen Datei gehört nicht zu einem der folgenden anerkannten Typen: Microsoft-Bibliothek, Intel-Bibliothek, Microsoft-Objekt, Xenix-Archiv.

U4150 *Modulname* : `Erneute Moduldefinition wird ignoriert`

Ein Modul wurde festgelegt, um in eine Bibliothek eingefügt zu werden, aber ein Modul desselben Namens existiert bereits in der Bibliothek. Oder ein Modul mit dem gleichen Namen wurde mehr als einmal in der Bibliothek gefunden.

U4151 *Symbol* : `Symbol im Modul Modulname erneut definiert; Redefinition ignoriert`

Das angegebene Symbol wird in mehr als einem Modul definiert.

U4153 *Zahl* : `Seitengröße zu klein; ignoriert`

Der mit der Option /PAGESIZE angegebene Wert ist kleiner als 16.

U4155 *Modulname* : `Modul ist nicht in der Bibliothek`

Ein zu ersetzendes Modul ist nicht in der Bibliothek. LIB fügt das Modul der Bibliothek hinzu.

U4156 *Bibliotheksname* : `Angabe der Ausgabebibliothek ignoriert`

Eine Ausgabebibliothek ist zusätzlich zu einem neuen Bibliotheksnamen angegeben. Wenn zum Beispiel

```
LIB new.lib+one.obj,new.lst,new.lib
```

angegeben wird, während *new.lib* noch nicht existiert, tritt dieser Fehler auf.

U4157 `Unzureichender Speicherplatz; erweitertes Verzeichnis nicht erzeugt`

LIB konnte kein erweitertes Wortverzeichnis erzeugen. Die Bibliothek ist weiterhin gültig, aber der Linker kann den Vorteil des erweiterten Wortverzeichnises zum schnelleren Binden nicht ausnutzen.

U4158 `Interner Fehler; erweitertes Verzeichnis nicht erzeugt`

LIB konnte kein erweitertes Wortverzeichnis erzeugen. Die Bibliothek ist weiterhin gültig, aber der Linker kann den Vorteil des erweiterten Wortverzeichnises zum schnelleren Binden nicht ausnutzen.

Index

"(Anführungszeichen), Feldende 3.23
""(Leere Zeichenkette) 3.12
'(Apostroph), Einleiten von
Kommentaren xxii
$(Dollarzeichen), Suffix für
Zeichenkettentyp 4.2
*(Stern)
Befehlssymbol LIB G.28
Zeichenkette fester Länge mit AS
4.2-3
,(Komma)
Feldende 3.23
Variablenbegrenzer 3.9
+(Plus)
Operator, Zusammensetzen von
Zeichenketten 4.5
Zeichen, Befehlssymbol LIB G.27
-(Minuszeichen), Befehlssymbol LIB
G.27
-*(Minuszeichen - Stern),
Befehlssymbol LIB G.28
-+(Minuszeichen - Pluszeichen),
Befehlssymbol LIB G.27
/(Schrägstrich), Optionszeichen LINK
G.15
;(Semikolon), Befehlssymbol LIB 3.3,
3.10, G.26

A

,A Option G.6
BASICA A.1
$INCLUDE Datei F.2
Abfragecodes
zur Tastenverfolgung 6.13
Ablaufsteuerung, Strukturen zur
Definition 1.1
Einrückung xxii
Entscheidung 1.5
Ablaufsteuerung (*Fortsetzung*)
neue 1.1
Schleife 1.17
verwendet in BASIC 9.2
ABS Funktion 8.1
ABSOLUTE H.9
Absolute Koordinaten *Siehe*
Koordinaten, absolute
Abtastcodes
für die Tastatur D.2, D.4, D.6
zur Tastenverfolgung 6.14
Adreßgleiche Variablen 2.35-36
/AH Option B.16, G.6
ALIAS, DECLARE Anweisung,
verwenden in 8.7
Alphabetisierung von Zeichenketten 4.6
AND Operator 1.3
AND Option
mit Graphikanweisung PUT 5.57
Anführungszeichen ("), Feldende 3.23
Animation
Bildflimmern reduzieren 5.64
Bildschirmseiten 5.66
einfache Graphikanweisungen 5.53
GET und PUT 5.53, 5.60
PALETTE USING 5.36
Animationsmodus B.18
Antwortdatei
Beispiel G.10
LIB G.24
LINK G.7
Anweisung, Änderung erfordert A.2
Anweisungen zur Ablaufsteuerung,
Siehe auch einzelne
Anweisungsbezeichungen
CALL 8.2-3
CALL ABSOLUTE 8.3
CALLS 8.3
CHAIN 8.3
Anweisung (*Fortsetzung*)
DEF FN 8.8
DO...LOOP 8.9
FOR...NEXT 8.11
FUNCTION 8.12
GOSUB...RETURN 8.12
GOTO 8.13
IF...THEN...ELSE 8.13
ON...GOSUB 8.20
ON...GOTO 8.20
RETURN 8.25
SELECT CASE 8.27
WHILE...WEND 8.34
Anweisung Block 1.2
Anzeige des Speichers, PCOPY-
Anweisung 8.21
Anzeigeausdrücke, Grenzwerte C.2
Apostroph (')
eingeben xix
Kommentare einleiten xxii
Arkustangens, Funktion ATN 8.2
Argumente
Optionen LINK G.15
Parameter
Übereinstimmung mit 2.20
unterschieden von 2.11
übergeben
als Grenzwert C.2
als Referenz 2.26
als Wert 2.26-27
übergeben an SUB-Prozeduren 2.10
überprüfen auf korrekten Typ und
korrekte Anzahl 2.20
ASC-Funktion 4.3, 8.2, 9.13
ASCII
Dateien, gelesen als sequentielle
Dateien 3.22

ASCII (*Fortsetzung*)
Zeichencodes D.2, D.5, D.6
Argumente für die Funktion CHR$ 4.3, 8.2
festlegen mit der Funktion ASC 4.3, 8.2
Zeichen im Hauptspeicher speichern 4.2
Assemblersprache, Listings B.11
ATN-Funktion 8.2
Attribute
Bildschirmmodi, EGA und VGA 5.31-32, 5.35
STATIC B.24
zuweisen verschiedener Farben 5.35
Aufruf, Fehlermeldungen I.1
Ausdrücke
Boolesche 1.2
falsche 1.3
Liste, Ausgabe 3.3
Übergabe an Prozeduren 2.14
wahre 1.3
Zeichenkette *Siehe* Zeichenketten
Ausführbare Dateien
Kompakt H.14
packen G.18
Ausfüllen *Siehe* Ausmalen
Ausfüllen mit Mustern 5.40
Siehe auch Ausmalen
Ausgabe
Anweisungen
Siehe auch einzelne Funktionsbezeichnungen
BEEP 8.2
CLS 8.5
LPRINT 8.18
OUT 8.21
PRINT 8.23
PRINT# 8.23
PRINT USING 8.23
PRINT# USING 8.23
PUT 8.24
WRITE 8.35
WRITE# 8.35
Funktionen
Siehe auch einzelne Funktionsbezeichnungen
LPOS 8.18
POS 8.23
TAB 8.31
Zeilenbreite 8.34
Ausgabebereich 3.3, 3.27
Ausgabegeräte 3.45
Ausgaben
Text
Bildschirmausgabe 3.2, 3.45
Druckerausgabe 3.45
von Zahlen
negative 3.3
positive 3.3, 4.14
Ausmalen
außerhalb 5.38
Farben 5.37-39
innerhalb 5.38, 5.41
Muster
Beschreibung 5.40, 5.68, 5.81
farbige 5.47
Auswählen
Befehle, Unterschiede zwischen Versionen B.12
Optionen, Unterschiede zwischen Versionen B.12
Automatische Variablen 2.7, 2.36
AUX (Gerätename) G.8

B

.bas-Datei, Ausführung mit QuickBASIC A.3
BASIC
Fehlercodes 8.10
Laufzeitfehler I.1
reserviertes Wort E.1
BASICA
Kompatibilität A.1
QuickBASIC, umwandeln in A.1
/BATCH Option (LINK) G.17
Baud-Rate 3.47
BC-Befehl
Befehlszeile, aufrufen aus G.3
Dateiname, Verwendung G.4
Optionen
(Liste) G.6
/A G.6
/AH B.16, G.6
/C G.6
/D B.15, G.6
/E B.15, G.6
/MBF B.4, B.6, G.6
/O G.6
/R B.16, G.6
/S B.16, G.6
/V B.16, G.7
/W 6.25, B.16, G.7
/X 6.25, B.16, G.7
/ZD G.7
/ZI G.7
BC-Befehl (*Fortsetzung*)
neue B.16
nicht verwendete B.15
Versionen, Unterschiede zwischen B.16
bc.exe G.2
Bearbeitung, Unterschiede zwischen Versionen B.15
BEEP-Anweisung 8.2
Befehl Bibliothek erstellen H.5-6
Befehl Hauptmodul setzen, Unterschiede zwischen Versionen B.13
Befehle
BC *Siehe* BC-Befehl
Linker *Siehe* LINK
Versionen, Unterschiede B.12
Befehle QB *Siehe* QB-Befehl
Befehlszeile
Quick-Bibliothek, erstellen aus H.10
Übergabe eines BASIC-Programms an 8.5
Benannter COMMON-Block 2.34, 2.41
Benutzerbibliotheken B.17
Siehe auch Quick-Bibliotheken
Benutzerdefiniert
Datentypen 8.32
Ereignisse 8.20, 8.33
Typen B.4
Bewegen von Bildern mit PUT 5.6
Bibliotheken
Siehe auch Quick-Bibliotheken
Beschreibung H.1
LINK, Angabe für G.12
selbständige
Beschreibung H.1
definieren H.1-2
LIB-Befehlszeile G.27
Listing G.27
Module, Löschen von, Einfügen und Ersetzen G.27
Module, Trennen und Löschen G.27
parallele Bibliotheken, erstellen H.11
zusammenstellen G.27
Standard ignorieren G.12, G.18
Suchpfad G.12
Typen, verglichen H.1
Bilder
kopieren auf den Bildschirm mit PUT 5.56
speichern in den Hauptspeicher mit GET 5.53-54

Bildschirm
Anweisungen
Siehe auch einzelne Anweisungsbezeichnungen
CLS 8.5
COLOR 8.5
LOCATE 8.18
PCOPY 8.21
SCREEN 8.26
VIEW PRINT 8.34
WIDTH 8.34
Bildschirmauflösung, SCREEN-Anweisung 5.3
Funktionen
Siehe auch einzelne Funktionsbezeichnungen
CSRLIN 8.6
POS 8.23
SCREEN 8.26
Konfiguration
Graphikmodus 5.2
Textmodus 3.2
Bildschirmmodi 5.2-3
Bildschirmseiten 5.66
Binär
Dateizugriff
Direktzugriff verglichen mit 3.41
OPEN-Anweisungssyntax B.24
Versionen, Unterschiede zwischen B.9
Eingabe 3.30
Suche, Beispiel 3.58
Zahlen, umwandeln in hexadezimal 5.42-43
Binärzugriffsdatei
erstellen 3.19, 3.40
lesen 3.19, 3.40
öffnen 3.41
schreiben in 3.41
Binden aus DOS G.2
Bit-Ebenen 5.48
Bitweise Operatoren 1.3-4
Siehe auch Logische Operatoren
BLOAD-Anweisung 8.2, A.2
Bogen 5.14, 5.16
Bogenmaß 5.14
Boolesche
Ausdrücke
Definition 1.2
logische Operatoren 1.3
vergleichen von anderen Ausdrücken 1.2
Vergleichsoperatoren 1.2
Konstanten 1.4
brun45.lib, Linkerstandard G.10
BSAVE-Anweisung 8.2, A.2
BUILDLIB-Dienstprogramm B.17
B_OnExit-Routine H.12

C

/C Option (BC) G.6
CALL-Anweisung
Aufruf einer SUB-Prozedur 2.10, 2.22
BASIC-Prozeduren 8.2
Beschreibung 9.3
DECLARE, verwendet mit B.21
nicht-BASIC-Prozeduren 8.3
optionale Verwendung B.21
QuickBASIC/Interpreterunterschiede A.2
CALL ABSOLUTE-Anweisung 8.3, H.9
CALL INT86OLD-Anweisung 8.3, H.9
CALL INT86XOLD-Anweisung 8.3
CALL INTERRUPT-Anweisung 8.3, H.9
CALL INTERRUPTX-Anweisung 8.3
CALLS-Anweisung 8.3
CASE$-Funktion B.23
CASE-Klausel 8.27
Siehe auch SELECT CASE-Anweisung
CDBL-Funktion 8.3
CDECL, Einsatz in DECLARE-Anweisung 8.7
CGA
Bildschirmmodi, unterstützt von 5.30
wechseln der Palette 5.32
CHAIN-Anweisung
Beschreibung 2.40, 8.3, 9.5
QuickBASIC/Interpreterunterschiede A.2
CHDIR-Anweisung 8.4
CHR$-Funktion 4.3, 8.4, 9.13
CINT-Funktion 8.4
CIRCLE-Anweisung
Beschreibung 8.4, 9.14
Bögen 5.14, 5.16
Ellipsen 5.12
Kreise 5.11
Tortendiagramme 5.17
CLEAR-Anweisung 2.40, 8.4, B.21
CLNG-Funktion 8.4, B.21
CLOSE-Anweisung 3.22, 8.5, 9.8
CLS-Anweisung 8.5, B.21
CodeView Debugger, LINK-Option für G.22
/CODEVIEW-Option (LINK) G.22
COLOR-Anweisung
Beschreibung 8.5, 9.15, B.21
Hintergrundfarbe, Steuerung von 5.5, 5.32
Syntax in Bildschirmmodus 1 5.32
verändern der Palette mit 5.32
Vordergrundfarbe, Steuerung von 5.5, 5.32
COM-Anweisungen 8.5, G.10
COM-Geräte 3.45
COMMAND$-Funktion
Beschreibung 8.5
Grenzwerte C.3
COMMON-Anweisung
AS Klausel B.21
Beschreibung 9.4
$INCLUDE-Metabefehl, verwenden mit 2.41
QuickBASIC/Interpreterunterschiede A.2
SHARED-Attribut
Beschreibung 9.4
Definition 2.28
Variablen global machen
gemeinsam benutzen 2.30, 2.33, 2.41
Verketten von Programmen 8.5
COMMON-Block
benannt 2.34, 2.41
unbenannt 2.41
CON (Gerätebezeichnung) G.8
CONS 3.45
CONST-Anweisung 8.6, B.21
COS Funktion (Kosinus) 8.6
/CPARMAXALLOC-Option (LINK) G.22
CSNG-Funktion 8.6
CSRLIN-Funktion 8.6, 9.7
Cursor
graphischer 5.7, 5.29
Text
Definition 3.13
finden der Position von 3.14
positionieren mit LOCATE 3.14
verändern der Form von 3.15
CVD-Funktion 8.6
CVDMBF-Funktion 8.6, B.6, B.22
CVI-Funktion 8.6
CVL-Funktion 8.6, B.21
CVS-Funktion 8.6

CVSMBF-Funktion 8.6, B.6, B.22
CVtyp-Anweisung 3.33, 9.12

D

/D-Option (BC) B.15, G.6
Darstellungsfeld
  graphisches
    VIEW SCREEN, definieren mit 5.20
    VIEW, definieren mit 5.21
    Vorteile 5.21
  Text 3.6
DATA-Anweisung 8.7
DATE$-Anweisung 8.7
DATE$-Funktion 8.7
Datei
  Namen
    gültige Zeichen 3.20
    OPEN-Anweisung 3.20
  Nummern
    CLOSE, freigeben mit 3.21
    FREEFILE, erhalten mit 3.19
    freigeben mit CLOSE 3.21
    gültige Werte 3.19
    OPEN-Anweisung 3.19
  Zeiger 3.41-42
Datei, E/A
  Definition 3.17
  Geräte-E/A im Unterschied zu 3.45
Dateibehandlung
  Anweisungen
    Siehe auch einzelne Anweisungsbezeichnungen
    CHDIR 8.4
    CLOSE 8.5
    FIELD 8.11
    GET 8.12
    INPUT # 8.14
    KILL 8.15
    LOCK 8.18
    NAME 8.20
    OPEN 8.20
    RESET 8.25
    SEEK 8.27
    UNLOCK 8.18
  Funktionen
    Siehe auch einzelne Funktionsbezeichnungen
    EOF 8.9
    FILEATTR 8.11
Dateibehandlung,
  Funktionen (*Fortsetzung*)
    FREEFILE 8.12
    LOC 8.17
    LOF 8.18
    SEEK 8.27
Dateien
  ASCII-Format A.1
  Attribute 8.11
  Direktzugriff, Anweisungen und Funktionen
    Siehe auch einzelne Anweisungs- und Funktionbezeichnungen
    FIELD 8.11
    LSET 8.18
    PUT 8.24
    RSET 8.26
  Grenzwerte C.1
  $INCLUDE F.1-2
  Länge, LOF 8.18
  map (LINK) G.19, G.21
  sequentielle, Anweisungen und Funktionen
    Siehe auch einzelne Anweisungs- und Funktionsbezeichnungen
    INPUT # 8.14
    LINE INPUT # 8.16
    PRINT # 8.23
    WRITE # 8.35
  Versionen, Kompatibilität zu B.25
  zusätzliche, von QuickBASIC erzeugt H.10
Dateinamen
  Abschneiden 3.20-21
  Einschränkungen 3.20
  Unterscheiden zwischen Groß-und Kleinbuchstaben 3.21
Dateinamen, Erweiterungen G.5
Dateinamen, Konventionen
  BASIC-Programme 3.20
  BC-Befehl G.4
  DOS 3.20-21
  LINK G.11
  Quick-Bibliotheken H.9
Dateiumwandlung 8.6
Dateizugriff
  LOCK-Anweisung 8.18
  UNLOCK-Anweisung 8.18, 8.33
Dateizugriffsmodi
  APPEND 3.19, 3.26
  BINARY 3.19, 3.40
  INPUT 3.19, 3.25
Dateizugriffsmodi (*Fortsetzung*)
  OUTPUT 3.19
  RANDOM 3.19
Daten-Dateien
  Dateinummern 3.19
  Definition 3.17
  Direktzugriff 3.18
  erstellen 3.18
  hinzufügen von Datensätzen 3.26
  lesen 3.29-31
  öffnen 3.18
  Organisation 3.17
  schließen 3.21
  sequentiell 3.18
  verfolgen von Dateizugriffsfehlern 6.26
  Vorteile von 3.17
Daten-Dateipuffer 3.21
Datendateien
  öffnen C.2
Datenfeld-Grenze, Funktionen 2.18
Datenfelder
  dynamisches
    ERASE-Anweisung 8.10
    REDIM-Anweisung 8.24
  Elemente übergeben 2.16
  Format
    Argumentenliste 2.13
    FUNCTION-Anweisung 2.16
    Parameterliste 2.13, 2.16
    SHARED-Anweisung 2.28
    SUB-Anweisung 2.12, 2.15
  Funktion für untere Grenze 2.18
  Grenzwert C.1
  Indizes, Angabe
    Anzahl 8.8
    Maximalwert 8.8
    untere Grenze 8.20
  LBOUND-Funktion 8.15
  Prozeduren
    gemeinsam benutzen mit 2.28
    Übergabe an 2.13, 2.15
  Speicherzuweisung F.2
  statische, ERASE-Anweisung 8.10
  UBOUND-Funktion (für obere Grenze) 2.18, 8.32
  Variablen 8.8
  Zeilenreihenfolge, Option B.16
Datensätze
  Beschreibung 3.33
  Binärzugriffsdateien, schreiben 3.40

Datensätze (*Fortsetzung*)
  definieren 3.33
  Definition 3.17
  Direktzugriffsdatei
    ergänzen 3.36
    hinzufügen 3.35
    lesen 3.37
    schreiben 3.32
    speichern 3.32
  fester Länge 3.19, 3.37
  sequentielle Dateien
    hinzufügen 3.18, 3.26
    lesen 3.25
    speichern 3.22
  übergeben an Prozeduren 2.13, 2.18
  überschreiben existierender Daten 3.18
  variabler Länge 3.23, 8.18
Datensatznummern
  Grenzwerte C.2
  Indizierung in einer Direktzugriffsdatei 3.39
Datensatznummern, Indizierung in einer Direktzugriffsdatei 3.54
Datentypen
  Angabe 8.8
  TYPE-Anweisung 8.32
Datum und Zeit
  Anweisungen
    DATE$ 8.7
    TIME$ 8.31
  Funktionen
    DATE$ 8.7
    TIME$ 8.31
Debug-Option B.15
DECLARE-Anweisungen
  Argumente überprüfen mit 2.20
  AS-Klausel B.21
  Definition 8.7, 9.3
  nicht von QuickBASIC erzeugt 2.21
  plazieren in Include-Dateien 2.23
  Versionen, Unterschiede zwischen B.22
  wo erforderlich 2.21
DEF FN-Anweisung 8.8, 9.4
DEF FN-Funktionen
  FUNCTION-Prozedur verglichen mit 2.5
  lokale Variablen in 2.5
  verlassen von, Alternative 8.10
DEF SEG-Anweisung 8.8
DEFLNG-Anweisung B.22
DEFtyp-Anweisungen
  Definition 8.8
  QuickBASIC/Interpreterunterschiede A.2
Deklarieren von Datenfeldern, Grenzwerte C.1
Dienstprogramm, BUILDLIB B.17
DIM-Anweisung
  AS-Klausel B.21
  Beschreibung 8.8
  QuickBASIC/Interpreterunterschiede A.2
  SHARED-Attribute
    Beispiel, adreßgleiche Variablen 2.35
    Definition 2.28
    Variablen global machen 2.30
    wann verboten 2.8
  TO-Klausel B.22
Dimensionieren von Datenfelder, Grenzwerte C.1
Direktes Fenster, Grenzwerte C.3
Direktzugriff, verglichen mit Binärzugriff 3.41
Direktzugriffsdateien
  anordnen von Datensätzen in 3.32
  Anweisungen und Funktionen
    Siehe auch einzelne Anweisungs- und Funktionsbezeichnungen
    FIELD 8.11
    GET 8.12
    LSET 8.18
    PUT 8.24
  erstellen 3.19
  hinzufügen von Daten zu 3.35
  lesen von Sätzen aus 3.37
  öffnen 3.19, 3.32
  sequentielle Dateien verglichen mit 3.18, 3.32
  wie Zahlen gespeichert werden in 3.32
DO UNTIL-Anweisung 8.9
DO WHILE-Anweisung 8.9
DO...LOOP-Anweisung
  Ablaufsteuerung 8.9
  alternatives Verlassen von 1.29, 8.10
  Beschreibung 9.2, B.22
  Position des Schleifentests 1.29
  Schlüsselwort
    UNTIL 1.30
    WHILE 1.30
  Syntax 1.25
  WHILE...WEND, verglichen mit 1.25
Dokumentkonventionen xix
Dollarzeichen ($), Suffix für Typ Zeichenkette 4.2
DOS 8.3, 8.9, 8.28
DOS-Ebene, Befehle der xix
/DOSSEG-Optionen (LINK) G.22
DRAW-Anweisung
  Beschreibung 8.9, 9.14
  Drehen von Abbildungen 5.67
  QuickBASIC/ Interpreterunterschiede A.2
  VARPTR$-Funktion, verwenden 8.33
Drehen von Abbildungen mit DRAW 5.67
Dummes-Terminal, Definition 3.62
$DYNAMIC-Metabefehl F.2
Dynamische Datenfelder
  $DYNAMIC-Metabefehl F.2
  ERASE-Anweisung 8.10
  REDIM-Anweisung 8.24

E

E/A
  Dateien, von und auf 3.17
  Geräte, von und auf 3.45
  Ports 8.20
  Standard 3.1, 3.8
/E-Option
  BC 6.25, B.15, G.6
  QB B.15
EGA, verändern der Palette 5.32, 5.35
Einfügemodus, Unterschiede zwischen Versionen B.10
Eingabe/Ausgabe *Siehe* E/A
Eingabe nach Dateiende: Fehler 3.27
  Eingabe/Ausgabe *Siehe* E/A
Eingabeanweisungen
  Siehe auch einzelne Anweisungsbezeichnungen
  DATA 8.6
  INPUT 8.14
  INPUT# 8.14
  LINE INPUT 8.16
  LINE INPUT# 8.16
  READ 8.24
  RESTORE 8.25
  WAIT 8.34
Eingabefunktionen
  Siehe auch einzelne Anweisungsbezeichnungen
  COMMAND$ 8.5
  INKEY$ 8.13

Eingabefunktionen (*Fortsetzung*)
INP 8.13
INPUT$ 8.14
Eingabeliste 3.9
Eingabetaste, entspricht der Leertaste B.12
Ellipsen, zeichnen 5.12
ELSE-Klausel 1.6
ELSEIF-Klausel 1.8
END-Anweisung 8.9
END CASE-Klausel 8.27
END DEF-Anweisung 8.9
END FUNCTION-Anweisung 8.9
END IF-Anweisung 8.9
END SELECT-Anweisung 8.9
END SUB-Anweisung 8.9
END TYPE-Anweisung 8.9
Enhanced Graphics Adapter *Siehe* EGA
Entscheidungsstrukturen
Definition 1.5
IF...THEN...ELSE
Block 1.8
einzeilige 1.6
SELECT CASE 1.10
ENVIRON-Anweisung 8.9
ENVIRON$-Funktion 8.9
EOF-Funktion 3.26, 3.45, 3.63, 8.9, 9.9
EQV-Operator 1.3
ERASE-Anweisung 8.10
ERDEV-Funktion 9.16
ERDEV$-Funktion 8.10, 9.17
Ereignisbehandlungsanweisungen
siehe auch einzelne Anweisungsbezeichnungen
COM 8.5
Ereignis ON 8.19
KEY (n) 8.15
KEY LIST 8.15
KEY OFF 8.15
KEY ON 8.15
ON UEVENT 8.20
PEN ON, OFF und STOP 8.22
PLAY ON, OFF und STOP 8.22
STRIG 8.30
TIMER ON, OFF und STOP 8.32
UEVENT 8.33
Ereignisbehandlungsoption B.16
Ereignisbehandlungsroutine 6.9
Ereignisregistrierung im Unterschied zu Ereignisverfolgung 6.8
Ereignisverfolgung
Siehe auch jeweiliges Ereignis
aktivieren 6.10

Ereignisverfolgung (*Fortsetzung*)
ausschalten 6.10
aussetzen 6.10
Befehlszeilenoptionen 6.25
Beschreibung 6.8
Hintergrundmusik 6.17
mehrmodulige Programme 6.20
Registrierung im Unterschied zu 6.8
Routine zur Ereignisbehandlung 6.9
SUB-oder FUNCTION-Prozeduren 6.19
Syntax im Unterschied zur Syntax der Fehlerverfolgung 6.10
Tastenbetätigungen 6.12
Typen
COM 6.10
KEY 6.10
PEN 6.10
PLAY 6.10
STRIG 6.10
TIMER 6.10
verfolgbare Ereignisse 6.10
Zusammenfassung von Anweisungen und Funktionen 9.16
Ereignisverfolgung, Quick-Bibliothek H.7
ERL-Funktion 8.10, 9.16
ERR
Code I.2
Funktion 6.3, 6.26, 8.10, 9.16
ERROR-Anweisung 8.10, 9.17
Erweiterungen, Dateinamen G.5
/EXEPACK-Option (LINK) G.18
EXIT-Anweisung 8.10, B.22
EXIT DEF-Anweisung 8.10, 9.5, B.22
EXIT DO-Anweisung 8.10, 9.2 B.22
EXIT FOR-Anweisung 1.22, 8.10, 9.2, B.22
EXIT FUNCTION-Anweisung 8.10, 9.3, B.22
EXIT SUB-Anweisung 8.10, 9.3, B.22
EXP-Funktion 8.10

F

Fakultät 2.38
Falsche Ausdrücke 1.3, 3.26
Farbattribute *Siehe* Attribute
Farben
festlegen in Graphikanweisungen 5.31
Hintergrund, steuern 5.5, 5.32
PALETTE USING, verändern mit 5.35-36

Farben (*Fortsetzung*)
PALETTE, verändern mit 5.35
Relation mit Schärfe 5.31
Verwendung mit CGA 5.30
Vordergrund, steuern 5.5, 5.32
Farbgraphikadapter *Siehe* CGA
Fehlerbehandlungsanweisungen
Siehe auch einzelne Anweisungsbezeichnungen
ERDEV 8.10
ERR, ERL 8.10
ERROR 8.10
ON ERROR 8.19
RESUME 8.25
Fehlerbehandlungsroutine
Fehler mit ERR identifizieren 6.3
spezifizieren 6.2
Teile 6.2
Fehlerbeseitigung
/CODEVIEW (LINK) Option G.22
Versionen, Unterschiede zwischen B.18
Fehlercodes 6.3, 8.10, I.2
Fehlermeldungen
Aufruf I.1
Beschreibung I.2
Kompilierzeit I.1
Laufzeit I.1
LIB I.44
LINK I.34
Umlenkung B.11
Zahlen, Grenzwerte C.2
Fehlermeldungsfenster B.12
Fehlerverfolgung
aktivieren 6.2
Dateizugriffsfehler 6.26
ERROR-Anweisung 8.10
Fehler mit ERR identifizieren 6.3
Fehlerbehandlungsroutine 6.2-3
mehrmodulige Programme 6.20-22, 6.23
Quick-Bibliotheken 6.22
Syntax im Unterschied zur Syntax der Ereignisverfolgung 6.10
ungeeignete Bildschirmmodi 5.2
Zusammenfassung von Anweisungen und Funktionen 9.16
Felder
Begrenzung in sequentiellen Dateien 3.23
Datensätze für Direktzugriff 3.32
Definition 3.17
sequentielle Datensätze 3.22

Felder, LINE-Anweisung 5.6
Fenster
direkte Fenster, Grenzwerte C.3
Fehlermeldungen B.12
Versionen, Unterschiede zwischen B.12
Fettgedruckter Text, markiert Schlüsselwörter xix
FIELD-Anweisung
Beschreibung 8.11
definieren von Datensätzen für Direktzugriff 3.33
TYPE...END TYPE, im Unterschied zu 3.33
FILEATTR-Funktion 8.11, 9.9, B.22
FILES-Anweisung 8.11, 9.9
FIX-Funktion 8.11
FOR...NEXT-Anweisung 1.22, 8.11, 9.2
FOR...NEXT-Schleifen
Ausführung unterbrechen 1.22
Beschreibung 1.17
STEP-Schlüsselwort 1.18
Verlassen, Alternative 8.11
Verschachteln 1.20
Formalparameter, POS-Funktion 3.16
Formatieren von Textausgabe 3.5
Fractal 5.74
FRE-Anweisung 2.40
FRE-Funktion 8.11
FREEFILE-Funktion 3.19-20, 8.12, 9.9, B.22
FUNCTION-Anweisungen
AS-Klausel B.21
Beschreibung 8.12
nicht erlaubt in Include-Dateien 2.25, F.2
STATIC-Attribute B.24
FUNCTION-Prozeduren
aufrufen 2.9
Beschreibung 2.7, 9.3, B.22
DECLARE-Anweisungen 8.7
DEF FN-Funktion im Unterschied zu 2.5
Ereignisverfolgung in 6.19
Fehlerverfolgung in 6.19
Verlassen, Alternative 8.10
FUNCTION...END FUNCTION-Anweisungen *Siehe* FUNCTION-Prozeduren
Funktionen, benutzerdefiniert 8.8
Siehe einzelne Funktionsbezeichnungen
Funktionstasten, Verfolgung 6.12

G

Ganzzahlen
FIX-Funktion 8.11
Grenzwert C.1
Umwandeln in 8.4, 8.11, 8.14
Gemeinsam benutzte Variablen zwischen Modulen 8.5
Geraden, zeichnen 5.6
Geradengestaltung 5.10
Geräte
Anweisungsbehandlung
IOCTL 8.15
OPEN COM 8.20
OUT 8.21
WAIT 8.34
Bezeichnungen
COM 3.45
CONS 3.45
E/A-Modi, gültige 3.45
Funktionsbehandlung
IOCTL$ 8.14
LPOS 8.18
PEN 8.22
KYBD 3.45
LPT 3.45-46
SCRN 3.45-46
Geräte-E/A, vergl. mit Datei-E/A 3.45
Geräte-Status Information 8.10
GET-Anweisung
Beschreibung 8.12, 9.9
Datei-E/A
Binärzugriff 3.41
Direktzugriff 3.38-39, 3.54
Datensätze, benutzerdefiniert B.23
Graphiken
Animation 5.60, 9.15
Berechnung der Datenfeldgröße in 5.54
Bilder in den Speicher kopieren 5.53
Syntax 5.53
Gleitkomma, Genauigkeit innerhalb von QuickBASIC-Bibliotheken H.8
Globale Variablen
adreßgleiche Variablen 2.35
DEF FN-Funktion 2.5
FUNCTION-Prozedur 2.5
GOSUB-Routine 2.3
SHARED-Attribut 2.30
GOSUB-Anweisung 8.12, 9.5
GOSUB-Unterroutine
Nachteil der Verwendung 2.3
Unterschied zu SUB-Prozeduren 2.2
GOTO-Anweisung
Beschreibung 8.13
$INCLUDE-Metabefehl F.2
Grad
Umwandlung in Bogenmaß 5.15
Vergleich mit Bogenmaß 5.14
Graphiken
Anweisungen
Siehe auch einzelne Anweisungsbezeichnungen
BLOAD 8.2, A.2
BSAVE 8.2, A.2
CIRCLE 5.11, 8.4
COLOR 5.32, 8.5
DRAW 8.9
GET 8.12
LINE 5.6, 8.16
PAINT 8.21
PALETTE USING 8.21
PALETTE 8.21
PRESET 8.23
PSET 5.5, 8.23
PUT 5.60, 8.24
VIEW 8.33
WINDOW 8.34
Funktionen
Siehe auch einzelne Anweisungsbezeichnungen
PALETTE USING 5.35
PMAP 5.29, 8.22
POINT 5.29, 8.22
VIEW 5.21
WINDOW 5.24
Graphische Anweisungen und Funktionen, Zusammenfassung 9.14
Graphische Bildschirmmodi *Siehe* Bildschirmmodi
Graphischer Cursor *Siehe* Cursor, graphischer
Graphisches Darstellungsfeld *Siehe* Darstellungsfeld, graphisches
Grenzwerte, Dateigröße und Komplexität C.1
Groß- und Kleinschreibung
Bedeutung, LINK-Optionen G.15, G.22
unterscheiden zwischen 3.21
Groß-und Kleinschreibung ignorieren
LIB G.29
LINK G.22

Großbuchstaben
Dateinamen 3.21
umwandeln in Kleinbuchstaben 4.12-13
GW-BASIC A.1

H

/H-Option (QB) B.16
Hauptmodul 7.2
/HELP-Option (LINK) G.16
HEX$-Funktion 8.13
Hexadezimale Zahlen 5.42-43, 8.13
Hintergrundfarbe, Standard 5.5
Hintergrundmusik 8.22
Hohe Auflösung, Anzeige-Option (QB) B.16

I

IEEE-Format
Genauigkeit B.4
/MBF-Option, Verwendung mit alten Programmen B.6
umwandeln in B.4, B.6
Zahlen
Ausgabe B.5
Bereiche B.5
exponentielle Ausgabe B.5
Microsoft Binär, Unterschiede zum B.5
umwandeln aus 8.19
umwandeln in 8.6
IF...THEN....ELSE-Anweisung, Blockform 8.13
IF...THEN...ELSE-Anweisung
Beschreibung 8.13, 9.2
Blockform 1.8-10
einzeilige Form 1.6
SELECT CASE-Anweisung im Unterschied zu 1.10
/IGNORECASE-Option (LIB) G.29
IMP-Operator 1.3
Include-Dateien
COMMON 2.41
nicht erlaubte Anweisungen 2.25
Prozeduren
Deklarieren von 2.23
nicht erlaubte B.17
Verschachtelungs-Grenzwerte C.2
$INCLUDE-Metabefehl
Beschreibung F.1
COMMON 2.41
Einschränkungen F.2
Prozedurvereinbarungen 2.23, 2.25
Indizes
Datenfelder, Grenzwerte C.1
Höchstwert angeben 8.8
Nummer angeben 8.8
obere Grenze für 8.32
untere Grenze angeben 8.20
/INFORMATION-Option (LINK) G.17
INKEY$-Funktion 3.12, 8.13, 9.6
INP-Funktion 8.13
INPUT-Anweisung
Anfrage 3.10
Beispiel 3.48
Beschreibung 8.14, 9.6
Definition 3.9
Fehlermeldung, ungültige Eingabe 3.10
FIELD-Anweisung 8.11
Format der Variablenliste 3.9
LINE INPUT im Unterschied zu 3.10
Unterdrücken von Wagenrücklauf-Zeilenvorschub-Folge nach Eingabe 3.11
INPUT#-Anweisung
Beispiel 3.25-26
Beschreibung 8.14, 9.8
INPUT$ im Unterschied zu 3.30
LINE INPUT# im Unterschied zu 3.29
INPUT$-Funktion
Beispiel 3.48
Beschreibung 8.14, 9.6, 9.9
Datenübertragung per Modem 3.63
INPUT# im Unterschied zu 3.30
Lesen von Daten
Dateien 3.30
Datenübertragungsgerät 348
Standardeingabe 3.12, 3.30
LINE INPUT# im Unterschied zu 3.30
INSTR-Funktion 4.7, 4.15, 8.14
INT-Funktion 8.14
INT86, INT86X, Ersatz
CALL INT86OLD-Anweisungen 8.3
CALL INTERRUPT-Anweisungen 8.3
INT86OLD H.9
Interpretiertes BASIC, Kompatibilität A.1
INTERRUPT H.9
Interrupt-Unterstützungsroutinen H.9
IOCTL$-Funktionen 8.14
IOCTL-Anweisung 8.15

J

Joystick 8.29-30

K

Kartesische Koordinaten *Siehe* Logische Koordinaten
Kein erweitertes Wortverzeichnis, Bibliothek G.30
KEY(n)-Anweisungen 8.15
KEY LIST-Anweisung 8.15
KEY OFF-Anweisung 8.15
KEY ON-Anweisung 8.15
KILL-Anweisung 8.15, 9.9
Kleinbuchstaben
Dateinamen 3.21
übertragen in Großbuchstaben 4.12-13
Komma (,)
Feldende 3.23
Variablenbegrenzer 3.9
Kommentare
eingeleitet mit Apostroph xxii
REM-Anweisung 8.24
Kompatibilität
BASICA und GW-BASIC A.1
Versionen, Dateien B.25
Kompilieren aus DOS G.2
Kompilieroptionen B.16
Kompilierzeit-Fehlermeldungen I.1
Komplexe Zahlen, Definition 5.74
Konstanten
Eingabeliste 3.9
symbolische xxi, 8.6
Übergabe an Prozeduren 2.14
Zeichenkette, literale und symbolische 4.2
Konventionen, typographische xix
Koordinaten
absolute
STEP, angeben mit 5.6
VIEW SCREEN, angeben mit 5.22

Koordinaten (*Fortsetzung*)
logische
definiert mit WINDOW 5.24
GET-Anweisung 5.55
übertragen in physikalische Koordinaten 5.29
physikalische
GET-Anweisung 5.55
Pixel positionieren 5.25
übertragen in logische Koordinaten 5.29
Pixel positionieren 5.3
relative
Definition 5.7
STEP 5.7
VIEW 5.21
Werte außerhalb des Darstellungsfeldes angeben 5.25
Kreise, zeichnen 5.11
Kursiver Text, anzeigen von Platzhaltern xx
KYBD 3.45

L

/L-Option (QB) H.7, H.9
Laden von Quick-Bibliotheken H.7
Lange Ganzzahlen
Grenzwerte C.1
umwandeln in 8.4, B.21
Vorteile der B.8
Laufzeit-Fehlermeldungen I.1
LBOUND-Funktion 2.18, 8.15
LCASE$-Funktion 4.13, 8.15, 9.11
Leere Zeichenkette ("") 3.12
Leerzeichen
abschneiden 4.6, 4.9-10
überspringen 3.5
LEFT$-Funktion 4.9, 8.16, 9.11
LEN-Funktion
Beispielprogrammanwendung 3.37, 4.15
Beschreibung 3.34, 8.16, 9.13, B.23
Verwendung von 4.3-4
LET-Anweisung 8.16
Letzter Punkt, auf den Bezug genommen wurde 5.6, 5.29
LIB
Siehe auch selbständige Bibliotheken
Antwortdatei G.24
Aufruf G.24
Befehlssymbol
Minuszeichen - Pluszeichen (+-) G.27
Pluszeichen (+) G.27
Stern (*) G.27-28
Beschreibung G.23
Bibliothekmodule G.27
Dateilisting G.26
Optionen
/IGNORECASE (/I) G.29
/NOEXTDICTIONARY (/NOE) G.30
/NOIGNORECASE (/NOI) G.30
/PAGESIZE (/P) G.30
Seitengröße, Angabe G.30
Wortverzeichnis, nicht erweitert G.30
Standardantworten G.26
Suchpfad G.12, H.8
Unterscheiden zwischen Groß- und Kleinschreibung G.29
Zusammenstellen von Bibliotheken G.27
Zusammenstellen von
lib.exe G.2
LIB-Fehlermeldungen I.44
LINE-Anweisung
Beispielanwendungen 5.68, 5.75, 5.81
Beschreibung 8.16, 9.14
Rechtecke und Geraden, zeichnen 5.8-9
Reihenfolge von Koordinatenpaaren 5.6
Syntax 5.6
LINE INPUT-Anweisung
Beschreibung 8.16, 9.7
INPUT im Unterschied zu 3.10
LINE INPUT#-Anweisung
Beschreibung 8.16, 9.9
INPUT# im Unterschied zu 3.29
INPUT$ im Unterschied zu 3.30
/LINENUMBERS-Option (LINK) G.21
LINK
Antwort-Datei G.7, G.10
Ausgabedatei, temporär G.13, G.16
Befehlszeile, Aufrufen aus G.7
Bibliotheken, Angabe G.12
Optionen *Siehe auch* Bibliotheken
/BATCH (/B) G.17
/CODEVIEW (/CO) G.22
/CPARMAXALLOC (/CP) G.22
/DOSSEG (/DO) G.22
/EXEPACK (/E) G.18
/HELP (/HE) G.16
/INFORMATION (/I) G.17
/LINENUMBERS (/LI) G.21
/MAP (/M) G.19
/NODEFAULTLIBRARYSEARCH (/NOD) G.12, G.18
/NOIGNORECASE (/NOI) G.22
/NOPACKCODE (/NOP) G.18
/PACKCODE (/PAC) G.21
/PAUSE (/PAU) G.16
/QUICKLIB (/Q) G.17
/SEGMENTS (/SE) G.19
/STACK (/ST) G.23
Abkürzungen G.15
Anzeigen G.16
Argumente, numerische G.15
BC ungeeignet für G.23
Befehlzeile, Reihenfolge in G.15
CodeView Debugger, debuggen mit G.22
Informationen anzeigen G.16
LINK-Umgebungsvariablen G.15
Linkeranfrage, vermeiden G.17
Map-Datei G.19
Paragraph, Zuweisung von Speicherplatz G.22
Quick-Bibliotheken, erstellen G.17
Segmente G.19
Segmente anordnen G.22
Standardbibliotheken, ignorieren G.12, G.18
Stapelgröße setzen G.23
Unterbrechen G.16
Unterscheiden zwischen Groß-und Kleinschreibung G.15, G.22
Zeilennummer anzeigen G.21
Quick-Bibliotheken, anderssprachige Routinen H.11
Standarde G.9-10

LINK-Fehlermeldungen I.34
LINK-Umgebungsvariable G.15
link.exe G.2
Linker *Siehe* LINK
Literale Konstanten, Zeichenkette 4.2
LOC-Funktion
  Beschreibung 8.17, 9.9
  Datenübertragung per Modem 3.63
  Geräte, mit 3.45
  SEEK im Unterschied zu 3.42
LOCATE-Anweisung
  Beispiel 3.49
  Beschreibung 8.18, 9.7
LOCK-Anweisung 8.18
LOF-Funktion
  Beispiel 3.54
  Beschreibung 8.18, 9.10
  Dateien 3.37
  Geräte 3.45
LOG-Funktion 8.18
Logische Koordinaten
  definieren mit WINDOW 5.25, 8.34
  physikalische Koordinaten
    MAP 8.22
    übertragen in 5.29
Logische Operatoren 1.3
Lokale Variablen 2.36
Löschen von Daten, vermeiden 3.25
LPOS-Funktion 8.18
LPRINT-Anweisung
  Beschreibung 8.18
  SPC-Funktion 8.28
LPRINT USING-Anweisung 8.18
LPT-Geräte 3.45-46
LSET-Anweisung 3.34, 8.18, 9.12, B.23
LTRIM$-Funktion 9.11
  abschneiden führender Leerzeichen 4.6
  ausgeben von Zahlen ohne führende Leerzeichen 8.18, B.23
  Beschreibung
  Zeichenketten fester und variabler Länge 4.10

M

MAK-Dateien, Quick-Bibliotheken, verwenden mit H.7
Mandelbrot-Menge 5.74
Map-Dateien (LINK) G.19-21
Marken xxii
Markierungen einsetzen, Grenzwerte C.3
Mathematische Funktionen
  ABS 8.2
  ATN 8.2
  COS 8.6
  CVSMBF 8.6
  EXP 8.10
  LOG 8.18
  MKSMBF$, MKDMBF$ 8.19
  SIN 8.28
  SQR 8.29
  TAN 8.31
/MBF-Option
  BC B.4, B.6, G.6
  QB B.5-6, B.16
Mehrmodulige Programme
  Ereignisverfolgung 6.20
  Fehlerverfolgung 6.20-23
  gemeinsame Benutzung von Variablen 2.33, 7.6
  Größenbegrenzungen C.2
  Hauptmodul 7.2
  Laden 7.4
  Programmierstil 7.10
  Quick-Bibliotheken 7.8
  Versionen, Unterschiede zwischen B.10
  Vorteile 7.1
Menü Ausführen, Befehl Bibliothek erstellen H.5-6
Menübefehle, Unterschiede zwischen Versionen B.12
Metabefehle
  Siehe auch $INCLUDE-Metabefehl
  Beschreibung F.1
  $DYNAMIC F.2
  $INCLUDE F.1
  $STATIC F.2
Microsoft Binärformat
  IEEE-Format, Unterschiede aus B.5
  Zahlen 8.6, 8.19
MID$-Anweisung 4.14, 8.18, 9.12
MID$-Funktion 4.11, 4.14, 8.18, 9.11
Minimieren der Option Zeichenkettendaten B.16
MKD$-Funktion 8.18
MKDIR-Anweisung 8.18
MKDMBF$-Funktion B.6, B.23
MKI$-Funktion 8.18
MKL$-Funktion 8.18, B.23
MKS$-Funktion 8.18
MKSMBF$-Funktion 8.19, B.6, B.23
MKtyp-Anweisung 9.12
MKtyp$-Anweisung 3.33
MKtypMBF$-Anweisung 3.33
Modem, Datenübertragung mit 3.63
Modul-Ebenen-Code 7.2
Module *Siehe* Mehrmodulige Programme
Monochromer Bildschirmmodus 5.30
Musik
  Anweisungen 8.22
  Hintergrund 6.17, 8.22
Musikereignisverfolgung
  ON PLAY GOSUB-Anweisung 6.17
  PLAY ON-Anweisung 6.17
Muster
  farbig 5.48
  Formen ausfüllen 5.40, 5.68, 5.81
  Kachelgröße 5.40
  monochrom 5.41
Musterkacheln, Editieren auf dem Bildschirm 5.81

N

NAME-Anweisung 8.19, 9.9
Neue Funktionen zur Zeichenkettenbehandlung B.23
Neue Zeile. *Siehe* Wagenrücklauf-Zeilenvorschub-Folge
NEXT-Anweisung 8.11, B.25
nocom.obj-Datei G.13
/NODEFAULTLIBRARYSEARCH-Option (LINK) G.12, G.18
/NOEXTDICTIONARY-Option (LIB) G.30
/NOIGNORECASE-Option (LIB) G.30
/NOIGNORECASE-Option (LINK) G.22
/NOPACKCODE-Option (LINK) G.18
NOT-Operator 1.3-4
NUL (Gerätename) G.8
nul.map-Datei G.10
Numerische Funktionen
  Siehe auch einzelne Funktionsbezeichnungen
  CDBL 8.3
  CINT 8.4
  CLNG 8.4
  CSNG 8.6
  CVD 8.6
  CVI 8.6
  CVL 8.6
  CVS 8.6

Numerische Funktionen (*Fortsetzung*)
FIX 8.11
INT 8.14
RND 8.26
SGN 8.28
Numerische Umwandlungen
CVD-Funktion 8.6
CVI-Funktion 8.6
CVL-Funktion 8.6
CVS-Funktion 8.6
doppellte Genauigkeit 8.3
einfache Genauigkeit 8.3
Ganzzahl 8.4, 8.11, 8.14

O

/O-Option (BC) G.6
Objekt
Dateien B.25, G.27
Module G.27-28
OCT$-Funktion 8.19
Oktalkonversion 8.19
ON Ausdruck-Anweisung 1.16
ON Ereignis-Anweisungen 8.19, 9.17
ON Ereignis GOSUB-Anweisung 6.25
ON ERROR GOTO-Anweisung
Beispiel 6.26
Beschreibung 8.19, 9.16
Syntax 6.2
ON...GOSUB-Anweisung 8.20
ON...GOTO-Anweisung 8.20
ON PLAY GOSUB-Anweisung 6.17
ON UEVENT-Anweisung 8.20, B.23
OPEN-Anweisung
Beschreibung 8.20, 9.8, B.24
Dateinamen in 3.20
Dateinummern 3.19
Dateizugriffsmodi
APPEND 3.19, 3.26
BINARY 3.19, 3.41
INPUT 3.19, 3.26
OUTPUT 3.19, 3.24, 3.26
RANDOM 3.19
LEN-Klausel 3.34
OPEN COM-Anweisung 3.47, 3.63, 8.20
Operatoren
logische 1.3
vergleichende 1.2
vergleichende 4.6
OPTION BASE-Anweisung 2.8, 8.20
Optionen *Siehe Einträge für* BC-Befehl; LIB; LINK
OR-Operator 1.3, 5.57
OUT-Anweisung 8.21
Overlay-linker *Siehe* LINK

P

PAINT-Anweisung
Siehe auch Ausmalen
Ausfüllen von Formen
farbiges Innere 5.38
gemustertes Innere 5.40, 5.68, 5.81
Beschreibung 8.21, 9.15
Hintergrundargument 5.47
Musterkachel
farbig, Definition 5.48
monochrom, Definition 5.41
numerischer Ausdruck 5.38
Randargument 5.39, 5.46
Zeichenkettenausdruck 5.40
/PACKCODE-Option (LINK) G.21
/PAGESIZE-Option (LIB) G.30
PALETTE-Anweisung 5.35, 8.21, 9.15, B.21
PALETTE USING-Anweisung 5.35, 5.75, 8.21
Paletten
Bildschirmmodus 1 5.32
COLOR, verändern mit 5.32
PALETTE, verämdern mit 5.35
Paragraphenabstand G.22
Parameter
Argumente
im Unterschied zu 2.11
Übereinstimmung mit 2.12, 2.20
Format in DECLARE 2.20
Vereinbarung von Anzahl und Typ 2.20
Parameterliste, Definition 2.20
/PAUSE-Option (LINK) G.16
PCOPY-Anweisung 8.20, 9.15
PEEK-Funktion 8.21
PEN-Funktion 8.21
PEN ON-Anweisung 8.22
PEN OFF-Anweisung 8.22
PEN STOP-Anweisung 8.22
Pfadnamen, Grenzwerte C.2
Pfeiltasten *Siehe* RICHTUNGSTASTEN
Physikalische Koordinaten 5.29, 8.22
Pixel
Definition 5.3
positionieren mit Koordinaten 5.3
PRESET, zeichnen mit 5.5
PSET, zeichnen mit 5.5
Textcursor 3.15
PLAY-Anweisung
Beschreibung 8.22
Option für Hintergrundmusik 6.17
QuickBASIC/Interpreterunterschiede A.2
VARPTR$-Funktion, verwenden 8.33
PLAY-Funktion 8.22
PLAY OFF-Anweisung 8.22
PLAY ON-Anweisung 6.17, 8.22
PLAY STOP-Anweisung 8.22
Plus (+)
Operator, kombinieren von Zeichenketten 4.5
Zeichen, LIB-Befehlssymbol G.27
PMAP-Anweisung 5.75
PMAP-Funktion 5.29, 8.22, 9.15
POINT-Funktion 5.29, 8.22, 9.15
POKE-Anweisung 8.23
POS-Funktion 3.16, 3.49, 8.23, 9.7
Positive Zahlen, Ausgabe ohne führendes Leerzeichen 4.14
PRESET-Anweisung 9.14
Beschreibung 5.5, 8.23
verwenden der Farboption mit 5.5
PRESET-Option mit der Graphikanweisung PUT 5.57
PRINT-Anweisung
Beispiel 3.49
Beschreibung 3.3, 8.23, 9.6
SPC-Funktion 8.28
Text umbrechen 3.4
PRINT #-Anweisung
Begrezung von Datensatzfeldern 3.28
Beschreibung 8.23, 9.8
WRITE# im Unterschied zu 3.27
PRINT USING-Anweisung 3.5, 3.49, 8.23, 9.6
PRINT USING#-Anweisung 9.8
PRINT# USING-Anweisung 8.23
PRN (Gerätenamen) G.8
Programm
aussetzen 8.28
beenden 8.9
Programme, BASICA, GW-BASIC, Umwandlung in QuickBASIC A.1

Programmieren anhand mehrerer Module 7.10
Programmieren in verschiedenen Sprachen
ALIAS, Verwendung 8.7
CALL-, CALLS-Anweisung (nicht-BASIC) 8.3
CDECL, Verwendung 8.7
DECLARE-Anweisung (nicht-BASIC) 8.7
Quick-Bibliotheken 7.8, H.10
Programmierstil xxi
Programmierumgebung, Unterschiede zwischen Versionen B.12
ProKey, verwenden von QuickBASIC mit B.10
Prozeduren
Anweisungen
FUNCTION...END FUNCTION 2.7-8
SUB...END SUB 2.7, 2.10
Zusammenfassung 9.3-4
Argumente übergeben
als Referenz 2.26
als Wert 2.27
Argumente übergeben als Referenz 2.26
aufrufen 2.9, 7.5
Ausdrücke 2.14
automatische Variablen 2,7, 2.36
Beschreibung 8.12
bewegen 7.4
Bibiotheken, Benutzer 7.2
Datenfelder 2.13, 2.15
Datensätze, 2.13, 2.18
definieren 2.7
Format der
Argumentenliste 2.13
Parameterliste 2.7, 2.13
Grenzwerte C.2
Include-Dateien,deklarieren in 7.6
Konstanten 2.14
mehrmodulige Programme 2.21, 7.1
nicht erlaubte Anweisungen in 2.8
Quick-Bibliotheken 2.25
rekursive 2.38, 2.43
STATIC-Variablen 2.7, 2.36-7
Variablen
gemeinsam benutzen 7.6
globale 2.30
lokale 2.5, 2.36
Vorteile 2.2
Prozeduren - nur auf Modulebene 7.5
PSET-Anweisung
Beispiel 5.75
Beschreibung 5.5, 8.23, 9.14
Farboption 5.5
STEP-Option
PSET-Option 5.57
Punktierte Geraden, zeichnen 5.10
PUT-Anweisung
Beschreibung 5.60, 8.24, 9.8
Datei-E/A
Binärzugriff 3.39-41
Direktzugriff 3.35-37, 3.54
graphische
Animation 5.60, 9.15
Bilder auf den Bildschirm kopieren 5.56
Datensätze, benutzerdefiniert B.23
steuern der Überlagerung mit dem Hintergrund 5.57
Syntax 5.56
Pufferzuweisung, FIELD-Anweisung 8.11

Q

QB-Befehl
/AH-Option B.16
/H-Option B.16
/L-Option H.9
/MBF-Option B.6-8, B.16
/RUN-Option B.16, H.8
Versionen, Unterschiede zwischen B.15
qb.qlb
automatisches Laden H.9
Bibliothek H.7
qbibdru.bas H.3, H.6-7
Quadrate, zeichnen 5.18
Quelldateien
Format A.1
Versionen, Kompatibilität zwischen B.25
Querverweise, Datei-Liste G.27
Quick-Bibliotheken
anderssprachige Routinen H.4
ausführbare Dateien, Kompaktieren H.14
Benennung H.6, H.9
Beschreibung H.1
Quick-Bibliotheken (*Fortsetzung*)
Dateien
erzeugt von H.10
zu erstellende H.4
Endbenutzer, liefern an H.10
Erstellen
Beschreibung H.1
erforderte Dateien 7.9, H.4
LINK verwenden von G.17
QuickBASIC, innerhalb von H.5
von Befehlszeilen von H.10
Fehlerverfolgung in 6.22
gemischte Programmiersprachen 7.8
Gleitkomma, Genauigkeit H.8
Inhalt
Beschreibung 7.8-9, H.3
Lesen 3.42
Listing H.8
Kompatibilität zwischen Versionen B.25
Kompilierung 7.8
Laden H.7, H.9
.mak-Datei, Aktualisieren H.7
Objektcode, binden 7.9
Prozeduren mit Include-Dateien deklarieren 2.25
Speicheranforderungen H.14
Standard H.7-8
Suchpfade H.8
verwenden von 7.8
vorhergehende Bibliotheken aktualisieren H.4
Vorteile H.2
/QUICKLIB-Option (LINK) G.17

R

/R-Option (BC) B.16, G.6
RANDOMIZE-Anweisung 8.24
READ-Anweisung 8.24
Rechtecke, Festlegung von Koordinaten 5.8
REDIM-Anweisung
Beschreibung 8.24
SHARED-Attribut 2.28, 2.30, 9.4
Registrierung 6.8
Rekursive Prozeduren 2.38-39
Relative Koordinaten *Siehe* Koordinaten, relative
REM-Anweisung 8.24

RESET-Anweisung 8.25
RESTORE-Anweisung 6.5, 8.25
RESUME-Anweisung
  Beispiel 6.26
  Beschreibung 8.25, 9.16
  erforderte Compileroption 6.25
  QuickBASIC/Interpreterunterschiede A.3
  RESUME NEXT im Unterschied zu 6.5
RESUME NEXT-Anweisung
  Beispiel 6.26
  Beschreibung 8.25
  RESUME im Unterschied zu 6.5
RESUME NEXT-Option B.16
RETURN-Anweisung 8.25, 9.17
RICHTUNGSTASTEN, verfolgen 6.12
RIGHT$-Funktion 4.10, 8.25, 9.11
RMDIR-Anweisung 8.25
RND-Funktion 8.26
Rollen 3.7
Routine, B_OnExit H.12
RSET-Anweisung 3.34, 8.26, 9.12
RTRIM$-Funktion 4.6, 4.9, 8.26, 9.11
RUN-Anweisung
  Beschreibung 8.26, A.3
  schließen von Daten-Dateien 3.22
/RUN-Option (QB) B.16, H.8

S

/S-Option (BC) B.16, G.6
SADD-Funktion 8.26
SAVE-Anweisung (BASICA) A.1
Schleifenstrukturen
  Definition 1.17
  DO...LOOP 1.25
  FOR...NEXT 1.17
  WHILE...WEND 1.23
Schlüsselwörter, Format in Programmbeispielen xxi
Schrägstrich (/), Optionszeichen LINK G.15
Schriftbild
  Platzhalter xx
  Programm xix
  Schlüsselwörter xix
Schriftbild (*Fortsetzung*)
  Tastennamen xx
SCREEN-Anweisung
  Anpassen der Bildschirmauflösung mit 5.3
  Auswirkung auf das Seitenverhältnis 5.18
  Beispiel 5.68, 5.75
  Beschreibung 8.26, 9.14, B.21
  Bildschirmseite 5.66
  Zeilenzahl im Textmodus 3.6
SCREEN-Funktion 8.26
SCRN 3.45-46
SEEK-Anweisung 3.18, 3.42, 8.27, 9.10, B.24
SEEK-Funktion 3.42, 8.27, 9.10
Segmente G.19
  Listen, Mapdateien G.19
  Packen G.18
  Reihenfolge G.22
  Zugelassene Nummer G.19
Seiten *Siehe* Bildschirmseiten
Seitengröße, Bibliothek G.30
Seitenverhältnis
  Berechnung für beliebige Bildschirmgröße 5.20
  CIRCLE-Anweisung 5.12
  Definition 5.19
  Korrektur wegen 5.20
  Quadrate zeichnen 5.19
Selbständige Programme außerhalb von QuickBASIC erstellen G.2
SELECT CASE-Anweisung
  Beschreibung 8.27, 9.2
  CASE ELSE-Klausel 1.14
  CASE-Klausel 1.12
  END SELECT-Klausel 1.13
  Fehlerbehandlungsroutine 6.3
  IF...THEN...ELSE-Anweisung im Unterschied zu 1.10
  ON Ausdruck GOSUB im Unterschied zu 1.16
  Syntax 1.12
  Versionen, Unterschiede zwischen B.24
Separate Kompiliermethode *Siehe* BC-Befehl
Sequentielle Dateien
  Anweisungen und Funktionen
    Siehe auch einzelne Anweisungsbezeichnungen und Funktionsnamen
    INPUT# 8.14
    LINE INPUT# 8.16
    PRINT# 8.23
    WRITE# 8.35
  Datensätze 3.22, 3.25
  Direktzugriffsdateien, verglichen mit 3.18, 3.32
  erstellen 3.18
  Felder 3.22
  hinzufügen von Datensätzen an 3.26
  öffnen 3.18-19, 3.25
Serielle Datenübertragung
  Definition 3.47
  Eingabepuffer 3.47
Serieller Kanal, öffnen für E/A 3.47
SETMEM-Funktion 8.27, B.24
SGN-Funktion 8.28
SHARED-Anweisung 2.28, 8.28, 9.4, B.21
SHARED-Attribut
  COMMON 2.28, 2.30, 2.33
  DIM
    Beispiel 2.35
    Beschreibung 2.28, 9.4
    gemeinsam benutzen 2.30
    wann verboten 2.8
  REDIM 2.28, 2.30, 9.4
SHELL-Anweisung 8.28
SideKick, Verwenden von QuickBASIC mit B.10
SIN-Funktion 8.28
SLEEP-Anweisung 8.28, B.24
Sortieren, Beispiele 1.5, 1.31, 3.61
SOUND-Anweisung 8.28
SPACE$-Funktion 4.12, 8.28, 9.13
Spalten
  überspringen 3.6
  Verändern der Anzahl auf dem Bildschirm 3.6
SPC-Anweisung 3.5-6
SPC-Funktion 8.28, 9.7
Speicher
  Anforderungen, Quick-Bibliotheken H.14

Speicher (*Fortsetzung*)
verfügbarer Berechnungsspeicher H.14
Speicherfunktionen 8.21, 8.23
Speichern
Programme, ASCII-Format A.1
von Bildern mit GET 5.53-54
Speicherverwaltung
Anweisungen
Siehe auch einzelne Funktionsbezeichnungen
CLEAR 8.4
DEF SEG 8.8
ERASE 8.10
PCOPY 8.21
Funktionen
Siehe auch einzelne Funktionsbezeichnungen
FRE 8.11
SETMEN 8.27
Sprachunterschiede zwischen Versionen B.18
SQR-Funktion 8.29
/STACK-Option (LINK) G.23
Standard-E/A
Definition 3.1, 3.9
in BASIC verwendete Anweisungen 9.6
Standardausgabe 3.2
Standardeingabe
Definition 3.9
lesen 3.9-11
umlenken 3.9
Stapelgröße setzen G.23
Stapelgröße, Anpassung für rekursive Prozeduren 2.40
Stapelverarbeitungsdatei, Verwendung G.3
STATIC
Anweisung 2.36-37, 8.29, 9.4
Attribut B.24
implizit dimensionierte Datenfelder F.3
Speicherzuweisung für Datenfelder F.2
Variablen 2.36
Statische Datenfelder
ERASE-Anweisung 8.10
STATIC-Metabefehl F.2
STEP-Option 5.6
Stern(*)
Befehlssymbol LIB G.28
Stern(*) (*Fortsetzung*)
Zeichenkette fester Länge mit AS 4.2-3
STICK-Funktion 8.29
STOP-Anweisung 8.29
Stoppbedingungen, Grenzwerte C.3
Stoppbits 3.47
STR$-Funktion 4.13, 8.29, 9.12
STRIG-Funktion 8.30
STRIG(n)-Anweisungen 8.30
STRIG OFF-Anweisung 8.30
STRIG ON-Anweisung 8.30
STRING$-Funktion 4.12, 8.30, 9.13
SUB
Anweisung
AS-Klausel B.21
Beschreibung 8.30, 9.3
nicht erlaubt in Include-Dateien 2.25
STATIC-Attribut B.24
Prozeduren
CALL 2.10
DECLARE-Anweisungen 8.7
Einsetzen in Include-Dateien B.17
Ereignisverfolgung in 6.19
Fehlerverfolgung in 6.19
GOSUB im Unterschied zu 2.3
$INCLUDE-Metabefehl F.2
lokale Variablen in 2.36
Verlassen, Alternative 8.10
Suchen
binär 3.58
Zeichenketten 4.7
Suchpfade
Bibliotheken G.12
Quick-Bibliotheken H.8
SuperKey, verwenden von QuickBASIC mit B.10
SWAP-Anweisung 8.30
Symbolische Konstanten
CONST 8.6
Definition B.21
Format in Programmbeispielen xxi
Zeichenkette 4.2
Symboltabelle in Mapdateien G.20
Syntaxanmerkungen
Auswahlmöglichkeiten xx
optionale Größen xx
Platzhalter xx
Syntaxüberprüfungsbefehl, Unterschiede zwischen Versionen B.9
SYSTEM-Anweisung 8.31
Systemaufrufe 8.3

T

TAB-Anweisung 3.6, 3.49
TAB-Funktion 8.31, 9.7
Tastatur, Lesen von, Eingabe von 3.9
Tastaturabfragecodes D.2, D.4, D.6
Tasten
Bearbeitung, Unterschiede zwischen Versionen B.14
Eingabetaste B.12
Fehlerbeseitigung, Unterschiede zwischen Versionen B.18
Leertaste B.12
Richtungstasten, 6.12
Tastenverfolgung
aktivieren 6.12
Tasten
benutzerdefinierte Tasten 6.13-14
Funktionstasten 6.12
Richtungstasten 6.12
Text-Darstellungsfeld *Siehe* Darstellungsfeld
Textausgabe
Bildschirm 3.3
Umbruch 3.4
Textfelder, Grenzwerte C.2
Textumbruch 3.4
TIME$-Anweisung 8.31
TIME$-Funktion 8.31
TIMER-Funktion 8.31
TIMER OFF-Anweisung 8.32
TIMER ON-Anweisung 8.32
TIMER STOP-Anweisung 8.32
TMP-Umgebungsvariable, LINK verwendet von G.12
Tortendiagramme, zeichnen 5.17
Trigonometrische Funktionen
ATN 8.2
COS 8.6
SIN 8.28
TAN 8.31
TROFF-Anweisung 8.32
TRON-Anweisung 8.32
TYPE-Anweisung
Beschreibung 8.32
Versionen, Unterschiede zwischen B.25
TYPE-Befehl (DOS) 3.27
TYPE...END TYPE-Anweisung
Beispiel 3.54
Definieren von Datensätzen in Direktzugriffsdateien 3.33

TYPE...END TYPE-
Anweisung (*Fortsetzung*)
im Unterschied zu FIELD 3.33
Zeichenketten fester Länge in 4.3
Typographische Konventionen xix

U

Übergabe als Referenz 2.26
Übergabe als Wert
DEF FN, verwendet in 2.6
Definition 2.27
Überprüfen einzelner
Anweisungsoptionen B.16
Überprüfung
Linker G.15
Segmente G.19
Stapelgröße G.23
Überschreibemodus, Unterschiede
zwischen Versionen B.10
Überspringen
Leerstellen 3.5
Spalten 3.6
UBOUND-Funktion 2.18, 8.32
UCASE$-Funktion 4.13, 8.32, 9.12,
B.23
UEVENT-Anweisung 8.33, B.25
Umgebungs-Zeichenkette 8.9
Umgebungsvariablen
Beschreibung 8.9
LIB G.12
LINK G.15
TMP, verwendet von LINK G.12
Umgeschaltete Tasten, verfolgen 6.14
Umwandlung
BASICA und GW-BASIC
Programme A.1
Daten dateien B.6
IEEE-Format, Programm für B.6
Unbenannter COMMON-Block 2.41
UNLOCK-Anweisung 8.18, 8.33
Unterbrechen der Programmausführung,
FOR...NEXT-Anweisung 1.22
Unterprogramme
Siehe auch SUB-Prozeduren
CALL-, CALLS-Anweisungen
SUB-Anweisung 8.2-3, 8.30
Unterroutinen
Siehe auch GOSUB-Unterroutine
8.19, 8.25
Unterschiede zwischen Versionen
Dateikompatibilität B.25
Tabelle B.1

V

/V Compileroption
Beschreibung 6.25, G.7
Versionen, Unterschiede zwischen
B.16
VAL-Funktion 4.13-14, 8.33, 9.12
Variablen
automatische 2.7, 2.36
Datenfelder
Beschreibung 8.8
Elemente übergeben 2.17
Gesamtübergabe 2.16
Datentypen 8.8
einfache Übergabe 2.15
gemeinsam benutzen, Programme
2.41
globale
adreßgleiche Variablen 2.35
Beschreibung 2.3-4
Funktionsdefinitionen 8.30
gemeinsam 2.30
lokale
Beschreibung 2.36
mehrmodulige Programme 2.33, 7.6
Namen
Format in Programmbeispielen xxi
Grenzwerte C.1
Prozeduren
gemeinsam benutzen, alle
Preozeduren in einem Modul
2.28, 2.30
übergeben an 2.26-27
STATIC 2.7, 2.36-37
Typ deklarieren 2.15
Werte austauschen, SWAP 8.30
Zeichenkette 4.1, 8.16
VARPTR-Funktion
Beschreibung 8.33
Versionen, Unterschiede zwischen
B.25
VARPTR$-Funktion
Beschreibung 8.33
DRAW, verwenden mit A.2
PLAY, verwenden mit A.2
VARSEG-Funktion
Beschreibung 8.33
Versionen, Unterschiede zwischen
B.25
Vereinbarungen
Siehe auch einzelne
Anweisungsbezeichnungen
CONST-Anweisung 8.6

Vereinbarungen (*Fortsetzung*)
DECLARE-Anweisung (BASIC-
Prozeduren) 8.7
DECLARE-Anweisung (nicht-BASIC
Prozeduren) 8.7
DEFtyp-Anweisung 8.8
DIM-Anweisung 8.8
Verfolgung
Siehe auch Ereignisverfolgung,
Fehlerverfolgung
mehrmodulige Programme 6.20
vom Compiler erforderte
Befehlszeilenoptionen 6.25
Vergleichen von Zeichenketten
fester und variabler Länge 4.6
Vergleichsoperatoren, mit 4.6
Vergleichsoperatoren
Boolesche Ausdrücke 1.2
SELECT CASE-Anweisung 1.12
Vergleiche von Zeichenketten 4.6
Verkettung von Programmen,
verwendete Anweisungen
Siehe auch einzelne
Anweisungsbezeichnungen
CHAIN 8.3
COMMON 8.5
Verkettung, Definition 4.5
Verlassen, Funktionen und Prozeduren
9.3-5
Versionsunterschiede,
Dateikompatibilität B.25
Verzeichnisanweisungen
Siehe auch einzelne
Anweisungsbezeichnungen
CHDIR 8.4
FILES 8.11
MKDIR 8.18
NAME 8.19
RMDIR 8.25
VGA (Video Graphics Adapter), Palette
ändern 5.35
VIEW-Anweisung 5.21, 5.75, 5.81,
8.33, 9.15
VIEW PRINT-Anweisung 3.6-7, 8.34,
9.7
VIEW SCREEN-Anweisung 5.22
VM.TMP-Anweisung G.13
Vordergrundfarbe
Bildschirmmodus 1 5.32
Standard 5.5
Vorwärtsbezug, Problem des 2.20

W

/W-Option (BC)
  Beschreibung 6.25, G.7
  Versionen, Unterschiede zwischen B.16
Wagenrücklauf-Zeilenvorschub-Folge 3.22
Wahre Ausdrücke 1.3, 3.26
WAIT-Anweisung 8.34
WEND-Anweisung
  Beschreibung 8.34
  Versionen, Unterschiede zwischen B.25
WHILE-Anweisung 8.34
WHILE...WEND-Anweisung
  Beschreibung 9.2
  DO...LOOP verglichen mit 1.25
  Syntax 1.23
WIDTH-Anweisung
  Beschreibung 8.34, 9.6, 9.8, B.21, B.25
  Spalten, verändern der Anzahl an 3.6
  Zeilen, verändern der Anzahl an 3.6
WINDOW-Anweisung
  Beispiel 5.75
  Beschreibung 8.34, 9.15
  GET, Auswirkung 5.55
  Koordinaten 5.25
WINDOW SCREEN-Anweisung, WINDOW verglichen mit 5.25
Winkelmessungen 5.14
WordStar-Tasten, Ähnlichkeit B.14
WRITE-Anweisung 8.35
WRITE#-Anweisung
  Beschreibung 8.35, 9.8
  PRINT#, im Unterschied zu 3.27

X

/X-Option
  Beschreibung 6.25, G.7
  Unterschiede zwischen Versionen B.16
XOR-Operator 1.3
XOR-Option mit Graphikanweisung PUT 5.57

Z

Zahlen
  Ausgabe auf den Bildschirm 3.3, 4.14
  Darstellung als Zeichenketten 4.13
  positive, Ausgabe ohne führendes Leerzeichen 4.14
Zahlen (*Fortsetzung*)
  Speicherung in Direktzugriffsdateien 3.32
Zahlen doppelter Genauigkeit
  Größenbegrenzung C.1
  umwandeln in 8.3
Zahlen einfacher Genauigkeit
  Größenbegrenzung C.1
  umwandeln in 8.6
Zeichen
  Grenzwerte C.3
  wie sie gespeichert werden 4.2
Zeichenkette
  Anweisungen
    Siehe auch einzelne Anweisungensbezeichnungen
    LSET 8.18
    MID$ 8.18
    RSET 8.26
  Ausdrücke
    Begrezung in sequentiellen Dateien 3.27
    Definition 1.12
  Funktionen
    Siehe auch einzelne Funktionsbezeichnung
    ASC 8.2
    CHR$ 8.4
    DATE$ 8.7
    HEX$ 8.13
    INPUT$ 8.14
    INSTR 8.14
    LCASE$ 8.15
    LEFT$ 8.16
    LEN 8.16
    LTRIM$ 8.18
    MID$ 8.18
    RIGHT$ 8.25
    RTRIM$ 8.26
    SADD 8.26
    SPACE$ 8.28
    STR$ 8.29
    STRING$ 8.30
    UCASE$ 8.32
    VAL 8.33
  Variablen 4.1, 8.16
  Verarbeitung *Siehe* Zeichenketten
Zeichenketten
  alphabetisieren 4.6
  Ausdrücke 4.1
  definiert 4.1
  ersetzen 4.14
  erzeugen 4.12
Zeichenketten (*Fortsetzung*)
  fester Länge
    AS STRING 4.2
    Datensatzelemente 4.3
    variabler Länge im Unterschied zu 4.7
  Grenzwerte C.1-2
  Konstanten 4.2
  Leerzeichen entfernen
    linke Seite 4.9
    rechte Seite 4.10
  suchen nach einer Teilzeichenkette 4.7
  Variablen 4.2
  variabler Länge
    AS STRING 4.2
    definiert 4.3
    fester Länge verglichen mit 4.7
    maximale Größe 4.3
  vergleichen 1.3, 4.6
  Zeichen ausgrenzen von
    der linken Seite 4.9
    der Mitte 4.11
    der rechten Seite 4.10
    Zusammenfassung der Anweisungen und Funktionen 9.11
  zusammensetzen 4.5
Zeichenketten fester Länge
  Datensatzelemente 4.3
  Parameterliste 2.13
  selbständige Variablen 4.3
  variable Länge, verglichen mit 4.7
Zeichenketten variabler Länge
  definiert 4.3
  GET-Anweisung 3.41
  maximale Größe 4.3
  PUT-Anweisung 3.41
  verglichen mit fester Länge 4.7
Zeilen, verändern der Anzahl an 3.6
Zeilendrucker 8.18
Zeilenende-Markierung 1.34
Zeitberechnungs-Funktion 8.31
/ZD-Option (BC) G.7
/ZI-Option (BC) G.7
Zufallszahlen 8.24, 8.26
Zusätzliche Dateien erzeugt von QuickBASIC H.10
Zuweisungsanweisungen 8.16
  Anführungszeichen, Feldende 3.23
  Dummes - Terminal, Definition 3.62
  Leere Zeichenkette 3.12

Microsoft®

1288 Artikelnr. 06027

Microsoft®

1288 Artikelnr. 06027

Microsoft QuickBASIC

Programmieren in BASIC

Microsoft® QuickBASIC

Programmieren in BASIC

Microsoft®